TRS-80® Model 4D
Disk System Owner's Manual

TANDY®

TERMS AND CONDITIONS OF SALE AND LICENSE OF TANDY COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND RADIO SHACK FRANCHISEES OR DEALERS AT THEIR AUTHORIZED LOCATIONS

LIMITED WARRANTY

CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this computer hardware purchased (the "Equipment"), and any copies of software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. **This warranty is only applicable to purchases of Tandy Equipment by the original customer from Radio Shack company-owned computer centers, retail stores, and Radio Shack franchisees and dealers at their authorized locations.** The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or a participating Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK MAKES NO EXPRESS WARRANTIES, AND ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE IS LIMITED IN ITS DURATION TO THE DURATION OF THE WRITTEN LIMITED WARRANTIES SET FORTH HEREIN.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

LIMITATION OF LIABILITY

A. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE." IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE."**

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the TANDY Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby Radio Shack sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by Radio Shack.

STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

SERVICE POLICY

Radio Shack's nationwide network of service facilities provides quick, convenient, and reliable repair services for all of its computer products, in most instances. Warranty service will be performed in accordance with Radio Shack's Limited Warranty. Non-warranty service will be provided at reasonable parts and labor costs.

Because of the sensitivity of computer products, and the problems which can result from improper servicing, the following limitations also apply to the services offered by Radio Shack:

1. If any of the warranty seals on any Radio Shack and Tandy computer products it sells are broken, Radio Shack reserves the right to refuse to service the product or to void any remaining warranty on the product.
2. If any computer product purchased from Radio Shack has been modified so that it is not within manufacturer's specifications, including but not limited to, the installation of any non-Radio Shack parts, components, or replacement boards, then Radio Shack reserves the right to refuse to service the product, void any remaining warranty, remove and replace any non-Radio Shack part found in the product, and perform whatever modifications are necessary to return the product to original factory manufacturer's specifications.
3. The cost for the labor and parts required to return the computer product to original manufacturer's specifications will be charged to the customer in addition to the normal repair charge.

TANDY® TRS-80® Model 4D Disk System Owner's Manual

The FCC Wants You to Know

This equipment generates and uses radio frequency energy. If not installed and used properly, that is, in strict accordance with the manufacturer's instructions, it may cause interference to radio and television reception.

It has been type tested and found to comply with the limits for a Class B computing device in accordance with the specifications in Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference in a residential installation. However, there is no guarantee that interference will not occur in a particular installation.

If this equipment does cause interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

- Reorient the receiving antenna
- Relocate the computer with respect to the receiver
- Move the computer away from the receiver
- Plug the computer into a different outlet so that computer and receiver are on different branch circuits

If necessary, you should consult the dealer or an experienced radio/television technician for additional suggestions. You may find the following booklet prepared by the Federal Communications Commission helpful: *How to Identify and Resolve Radio-TV Interference Problems*.

This booklet is available from the US Government Printing Office, Washington, DC 20402, Stock No. 004-000-00345-4.

Warning

This equipment has been certified to comply with the limits for a Class B computing device, pursuant to Subpart J of Part 15 of FCC Rules. Only peripherals (computer input/output devices, terminals, printers, etc.) certified to comply with the Class B limits may be attached to this computer. Operation with non-certified peripherals is likely to result in interference to radio and TV reception.

10 9 8 7 6 5 4 3 2 1

Introduction

About Your Computer

Congratulations on the purchase of your Tandy® TRS-80® Model 4D Microcomputer System. Your new computer is compact and perfect for business needs, as well as personal use. You will find it to be a valuable tool that will save you work as well as give you hours of enjoyment. Its basic features include:

- 64K (65536 characters) of random access memory, expandable to 128K.
- High-speed Z-80A microprocessor, the "brains" of the computer.
- Upper and lower case text display of 80 characters by 24 lines or 64 by 16 (software selectable).
- Compatibility with the Radio Shack® Model III Software Library.
- Two built-in drives that let you use single- or double-sided, double-density floppy disks.
- Sound generation.
- 71-key console keyboard that includes three function keys and a numeric keypad.
- Built-in printer interface.

As your needs grow — you can expand your computer to include hard disks, external floppy disk drives, high-resolution graphics, printers, RS-232-C communications, and more.

About This Manual

This manual shows how you can use your disk system to:

- Store, retrieve or manipulate information on disk (using TRSDOS®).
- Write programs for your computer (using BASIC).

Your computer's operating system is TRSDOS Version 6.2.1 (or any later version of TRSDOS). Throughout this manual, we refer to this system simply as TRSDOS or TRSDOS Version 6.

In the *Introduction To Your Disk System* manual, we covered all the essential information to get you started. As you learn more about TRSDOS and programming, you can take advantage of its many features explained in this manual.

Since this is a reference manual, you don't have to read it from front to back. If you are a programmer, you'll find a lot of useful information in this manual. If you are an advanced programmer, you'll find additional technical information available in *The Model4/4P Technical Reference Manual* (Cat. No. 26-2119), which is available at your local Radio Shack store.

Part I/ TRSDOS

Section I/ Using TRSDOS describes how to start up and use TRSDOS.

Section II/ TRSDOS Commands contains a detailed explanation of all TRSDOS Version 6 commands and utilities.

Part II/ Model 4 BASIC

Section III/ Operations explains how to start up and operate BASIC.

Section IV/ BASIC Language introduces some general concepts about BASIC, and tells how to store data on disks. Chapter 7 of this section contains a detailed explanation of all Model 4 BASIC statements and functions.

Table of Contents

Part I/ TRSDOS Version 6

Section I/ Using TRSDOS

Section I/ Using TRSDOS

How The Computer Uses TRSDOS

Whenever you are using a program which runs under TRSDOS, your computer will, from time to time, need to reference TRSDOS. It always looks for TRSDOS on Drive 0.

For this reason, you must at all times have TRSDOS in Drive 0.

TRSDOS Notations

For clarity and brevity, we use some special notations and type styles in this section.

CAPITALS and punctuation
indicate material that you must enter exactly as it appears or material that you see on your computer's video display.

(KEYBOARD CHARACTER)
indicates key you press.

italics
represents words, letters, characters, or values that you supply.

TRSDOS Terms

Below is a listing of terms which we use frequently in this section. The italicized words represent variable information which you must supply.

command	represents the TRSDOS command you want to execute. *command* can be in upper or lowercase letters.
(*parameters*)	is a list of one or more values that may be needed by the command. Some commands have no parameters. Most parameters are optional. Brackets [ ] around any word in a command line indicate that it is optional.
number	any decimal or hexadecimal numeric expression. Hexadecimal expressions must be in the X'*nn*' format for bytes or X'*nnnn*' format for words, where *nn* is the hexadecimal value.
filespec	is a standard TRSDOS file specification having the general form: *filename/ext.password:drive*
devspec	is (1) one of six standard TRSDOS device specifications, or (2) a user created device specification having the general form: **two-letter abbreviation*
diskette	refers exclusively to a floppy diskette.

disk	refers to a floppy diskette, hard disk, or Memdisk.
disk ID	refers to the disk NAME, creation date, and Master Password.
I/O	refers to a transfer of data (Input/Output).

TRSDOS Abbreviations

You can abbreviate a *parameter* to its first letter (unless otherwise stated in the *command* explanation). You can also abbreviate YES to Y and NO to N.

Loading TRSDOS

When you install and power up your system, you'll see the TRSDOS start-up logo. This means you're in the TRSDOS Version 6 Operating System. You then need to enter the current date in the form *mm/dd/yy*. For example, for June, 18, 1985, type:

```
06/18/85 (ENTER)
```

The system displays the date in expanded form (for example, Tue, Jun 18, 1985).

You may use any of the ASCII characters in the range 32 (X'20') through 39 (X'27'), 41 (X'29') through 47 (X'2F'), and 58 (X'3A') to separate the month and day and day and year. See Appendix C for a complete list of ASCII character codes.

TRSDOS Ready

Whenever you see the TRSDOS Ready prompt you know that you are communicating with TRSDOS — not COBOL, PAYROLL, or any of your application programs. Communicating with TRSDOS allows you to do one of these operations:

- execute a TRSDOS system command or utility program
- execute an application program

When an error occurs, it comes from one of three places: an application program that you are running, TRSDOS, or a particular TRSDOS library command that you are using. If you think the error comes from an application program, look for the explanation in the manual that comes with the program. If you think it comes from TRSDOS, see Appendix D of this manual. If you can't find the explanation there, check the individual command in Section II.

Executing A Command

You can execute a TRSDOS system command whenever you see the TRSDOS Ready prompt. The command you type can consist of up to 79 upper or lower case characters. You must complete the command by pressing (ENTER).

For example, if you want to see the TRSDOS system commands, type:

```
LIB (ENTER)
```

TRSDOS displays a list of all the available system commands and returns to TRSDOS Ready:

```
Library <A>
  Append  Cat    Cls    Copy    Device  Dir     Do
  Filter  Lib    Link   List    Load    Memory  Remove
  Rename  Reset  Route  Run     Set     Tof

Library <B>
  Attrib  Auto   Build  Create  Date    Debug   Dump
  Free    Purge  Time   Verify

Library <C>
  Forms   Setcom Setki  Spool   Sysgen  System
```

If you want this display to print on the printer, type (CTRL)(:). Whenever you press this key sequence, what is displayed on the screen is printed on the printer.

If you type (CTRL) (R) at the TRSDOS Ready prompt, TRSDOS redisplays the last executed TRSDOS command and executes it again.

If you type (*) (ENTER) at the TRSDOS Ready prompt, TRSDOS executes an Immediate Execution Program (IEP) that is stored in the SYS13/SYS file. See Appendix I, "Immediate Execution Program" for information on implementing an IEP.

Disk Files

You can keep a record of anything you type into your computer by storing it on disk in a "disk file." For example, a disk file can contain a program, a collection of data, or a project report. But whatever it is, if you want to keep it permanently, you'll have to store it in a disk file.

When the computer stores the file, it records the name of the file and its disk location in a special place on the disk called the disk's directory. Whenever you want to access the file, the computer can immediately find its location by using this directory.

Filespec

Whenever you create a disk file, you need to give it a name. This name is just one part of a file specification — filespec, for short. The filespec is the standard TRSDOS format you'll use every time you reference your file:

    filename/ext.password:drive

filename
The name of your file can be anything you like, as long as it is one to eight alphanumeric characters, the first of which must be a letter. (The only names you cannot use are TO, ON, USING and OVER.)

/extension
If you want to further identify your file, you can give it a second name by adding an extension. An extension (indicated by /*ext* on our filespec) is a sequence of one to three alphanumeric characters (the first of which must be a letter) with a preceding slash (/).

You can use an extension to provide additional information on a file, or you can use an extension to indicate the type of file you have.

Some TRSDOS commands require that you specify an extension; if you don't, the command assumes a default extension. To override the default for a file that doesn't have an extension, supply the filename followed by a slash (/).

.password
A password protects a file by limiting access to it. You can accomplish this protection via a password either when you create the file or with the ATTRIB command.

A password is a sequence of up to eight alphanumeric characters, the first of which must be a letter.

:drive
Often when you're using your computer, you'll have more than one disk drive in use. You can speed up the file access time by specifying the drive the desired file is on.

If you omit a drive number on the filespec, your computer automatically starts looking for the file on all available drives, beginning with Drive 0.

Here are some examples of valid TRSDOS filespecs:

    DOPROG.OPEN
    CLR/BAS:1
    MOD16:4
    STL12/TXT.ARCH:1
    GAME1
    THESIS/OLD:2
    CONTEMP:3

You cannot use TO, ON, OVER, or USING as TRSDOS filespecs.

Partspecs

Certain system commands and utilities allow you to specify a collection of files by using a "partspec." A partspec is used with a "wildcard" mask ($). When you use a wildcard in a partspec, it represents a wildcard field and means "any character." For example, suppose the following files exist on a disk in Drive 1:

```
A                       ACORN
ADVANCE/DAT             ADVISE/DAT
BILLING/CMD             BILLING
BILLING/BAK             BILLING/DAT
```

If you issue the command:

```
CAT A:1 (ENTER)
```

TRSDOS displays these files:

```
A                       ACORN
ADVANCE/DAT             ADVISE/DAT
```

All files on the disk that begin with the letter A are displayed because when you specify a partspec, TRSDOS treats the command as "CAT of all files that begin with 'A'."

If you issue the command:

```
CAT BILLING:1 (ENTER)
```

TRSDOS displays:

```
BILLING                 BILLING/CMD
BILLING/DAT             BILLING/BAK
```

Because you did not specify an extension, TRSDOS assumed that all extensions are acceptable.

If you issue the command:

```
CAT /$A:1 (ENTER)
```

TRSDOS displays the files on the disk which have an A as the second character in their extension:

```
ADVANCE/DAT             ADVISE/DAT
BILLING/DAT             BILLING/BAK
```

Because you did not specify a filename, TRSDOS assumed that all filenames are acceptable.

A wildcard character must always have at least one character to its right. The following partspecs select the same group of files:

```
A                       A$
A$$$                    A$/$
A$/$$$
```

Devices

There are two kinds of TRSDOS devices: physical and logical.

A physical device is a piece of your computer hardware: the video display, the keyboard, the printer, etc.

A logical device (devspec) is a connection between TRSDOS and a physical device.

TRSDOS lets you treat your devices independently, which means you can sometimes substitute a device for another one. You can also substitute a file for a device. See the LINK, ROUTE, and SET library commands.

Devspec

When you want to access a device, you use its device specification or *devspec*.

TRSDOS devices already have devspecs assigned to them. You assign devspecs to devices that you create. The devspec is a two-character abbreviation for the device. The first character must be a letter. The second character can be a letter or a number. An asterisk must precede the devspec.

Your original TRSDOS master diskette is configured with six devices. They are:

Devspec	Device
*KI	Keyboard Input
*DO	Display Output (Video)
*PR	Printer
*SI	Standard Input
*SO	Standard Output
*JL	Job Log

These are known as system devices.

Drivers And Filters

Each device is controlled by its own driver program, filter program, or both. You can change a device's I/O by manipulating its driver or filter program. For more information on drivers and filters see the SET and FILTER commands; see also Appendices I and K.

Section II/ TRSDOS Commands

Section II/ TRSDOS Commands

TRSDOS commands and utilities (typed in at the TRSDOS Ready level) perform a variety of helpful operations:

Diskette Handling commands allow you to prepare your blank diskettes for use or make copies of existing diskettes. Any time you use a blank diskette, you should use this command:

FORMAT

If you want to change the way your computer system starts up and initializes its parameters, you can use *Initialization* commands. For example, you can use the FORMS commands to set your printer's parameters; or you can use the AUTO command to set your computer to AUTOmatically perform a particular function at start-up. The Initialization commands are:

AUTO	SETKI
BOOT	SYSGEN
DATE	SYSTEM
FORMS	TIME
SETCOM	

You might find the *Auxiliary* commands helpful for such functions as seeing what is on your disk or simply seeing what system commands are available. They include:

CAT	LIST
CLS	LOG
DEVICE	MEMORY
DIR	SPOOL
DO	VERIFY
FREE	TOF
LIB	

The *File Handling* commands and utilities allow you to copy, rename, delete, or convert your disk files. These commands include:

APPEND	CREATE
ATTRIB	PATCH
BACKUP	PURGE
BUILD	REMOVE
COMM	RENAME
CONV	REPAIR
COPY	TAPE100

The *Device Handling* commands allow you to set, filter, route, or reset your devices. Be sure you have a good understanding of devices before you use these commands! These commands include:

FILTER	RESET
LINK	ROUTE
MEMDISK	SET

Machine Language File Handling commands create and execute machine language disk files. These commands include:

DEBUG	LOAD
DUMP	RUN

How to Use This Section

This section contains an alphabetic listing of all TRSDOS commands and utilities. The commands and utilities for advanced programmers are marked as "Advanced Programmer's Utilities" and "Advanced Programmer's Commands."

Commands

Commands are system operations that can be used at TRSDOS Ready.

To see a list of all library commands, use the LIB command. Type:

LIB (ENTER)

and the following list is displayed:

```
Library <A>

  Append  Cat     Cls     Copy    Device  Dir     Do
  Filter  Lib     Link    List    Load    Memory  Remove
  Rename  Reset   Route   Run     Set     Tof

Library <B>

  Attrib  Auto    Build   Create  Date    Debug   Dump
  Free    Purge   Time    Verify

Library <C>

  Forms   Setcom  Setki   Spool   Sysgen  System
```

Utilities

Utilities use some or all of user memory. They return to TRSDOS Ready; under most conditions you cannot use them effectively within programs.

The utilities are:

BACKUP	LOG
COMM	PATCH
CONV	REPAIR
FORMAT	

Entry Organization

Each entry in this section is identified as either a command or a utility.

The command's "syntax" is the first line you see after the keyword. Use it as your guide to type in a command. (See "Syntax" below for details.)

A description of the command or utility follows the syntax. This description tells you what the command or utility does. Next, the entry includes additional information on the parameters of the command. A command may require you to supply some values.

The definition also may offer several "options" that customize the command to your needs. These optional parameters increase the usefulness of the commands but are not necessary for normal operation. Values and options are discussed in the additional parameter information.

Finally, each entry gives examples of the command's use.

Syntax

The command's syntax tells you the format to use when typing the command.

For example, here is the syntax for the COPY command:

COPY ***source*** [**TO**] ***destination*** [**(*parameters*)**]

In the syntax, italicized words indicate values that you supply. Words or values that are enclosed in brackets are optional. Those that are **not** enclosed in brackets are required.

In this case, you must type the keyword, which is COPY, followed by the filespec of the file to be copied and the filespec you want assigned to the duplicate file.

For example:

```
COPY NEW/DAT:1 NEWDAT/ONE:2 (ENTER)
```

copies the Drive 1 file NEW/DAT to the diskette in Drive 2 and names the new file NEWDAT/ONE.

The COPY command offers four optional parameters: LRL, CLONE, ECHO, and X.

Suppose you need to assign a specific record length to the new file. In this case, include the LRL parameter, as in this example:

```
COPY NEW/DAT:1 TO NEWDAT/ONE:2 (LRL=128) (ENTER)
```

Notice that you do not type the brackets that indicate that TO is an optional parameter, but that you **do** type the parentheses around LRL=128.

A vertical bar (¦) in a syntax indicates an either/or situation. For example, DEBUG lets you turn the debugger either ON or OFF.

NOTE: You cannot use the word TO as a filespec.

APPEND

APPEND *source* [TO] *destination* [(*parameters*)]	**Command**

Appends the contents of the *source* onto the end of the contents of the *destination*. (The contents of the source file remain the same.)

You can use APPEND to combine two files on a disk.

The *source* is a valid TRSDOS filespec or input devspec, and *destination* is a valid TRSDOS filespec.

The parameters are:

ECHO echoes the characters to the screen when appending a device to a file.
STRIP backspaces the destination file one byte before the append begins.

APPEND is most useful for data files because you can use APPEND only with ASCII files. You cannot APPEND files that are in the "load module format." To APPEND BASIC programs you must save them in ASCII format by specifying the A option with the BASIC SAVE command.

If you omit an extension with the destination filespec, TRSDOS tries to find the destination filename with the same extension that the source filespec has. If you want to force TRSDOS not to include an extension with the destination filespec, include only the slash (/) and omit additional characters when specifying the destination filespec.

Some programs place an end-of-file marker at the end of a file. Use the STRIP parameter to remove this marker when you APPEND a file. If you omit the STRIP parameter, the program ignores the appended section of the file.

Examples

```
APPEND EAST/DAT:1 TO WEST/DAT:0 (ENTER)
```

adds the information in EAST/DAT on Drive 1 onto the end of the information in WEST/DAT on Drive 0.

```
APPEND EAST/DAT:1 TO WEST/DAT:0 (STRIP) (ENTER)
```

adds the contents of EAST/DAT to the end of the contents of WEST/DAT as in the previous example. However, APPEND removes the last byte of WEST/DAT before appending.

```
APPEND *KI TO WEST/DAT:0 (ENTER)
```

appends the information that you type on the keyboard to the end of WEST/DAT on Drive 0. Press (CTRL)(SHIFT)(@) (at the same time) to end the append.

```
APPEND *KI TO WEST/DAT:0 (ECHO) (ENTER)
```

displays what you are appending to WEST/DAT as you type it. Press (CTRL)(SHIFT)(@) (at the same time) to end the append.

Error Conditions

If the records in *source* and *destination* are not the same length, TRSDOS displays a "Files have different LRLs" error message. Use the DIR command to display the file's LRL. Use the COPY command to change the LRL of a file.

If you omit *destination*, TRSDOS displays a "Destination spec required" error message. Enter the command again, specifying *destination*.

If you omit *source*, TRSDOS displays a "File spec required" error message. Enter the command again, specifying *source*.

If you attempt to append a device to a device, TRSDOS displays a "File spec required" error message. You must specify a filespec as *destination* in an APPEND command.

If you omit the extension for destination filespec, TRSDOS tries to find the destination filename with the same extension that source filespec has. If TRSDOS cannot find that filespec, TRSDOS displays a "File not in directory" error message.

Sample Uses

Suppose you have two data files, PAYROLL/A and PAYROLL/B.

PAYROLL/A	PAYROLL/B
Atkins, W.R.	Lewis, G. E. ....
Baker, J. B.	Miller, L. O. ....
Chambers, C. P.	Peterson, B. ....
Dodson, M. W.	Rodriguez, F. ....
Kickamon, T. Y.	

You can combine the two files with the command:

```
APPEND PAYROLL/B TO PAYROLL/A (ENTER)
```

PAYROLL/A now looks like this:

```
        PAYROLL/A
Atkins, W. R.  ...............
Baker, J. B. .................
Chambers, C. P.  ............
Dodson, M. W. ..............
Kickamon, T. Y.  ............
Lewis, G. E.  ................
Miller, L. O. .................
Peterson, B.  ................
Rodriguez, F.  ...............
```

PAYROLL/B is unaffected. To see the APPENDed file, type LIST PAYROLL/A.

ATTRIB

ATTRIB *filespec* [(*parameters*)]	**Command**

ATTRIB [:*drive*] [(*disk parameters*)]	**Command**

Assigns protection passwords and attributes to a particular file or a group of files.

You can use ATTRIB to protect a file with passwords.

For filespec ATTRIBs, the parameters are:

USER="*password*" sets the user password to *password*. If this parameter is omitted, the user password remains the same. If USER= is specified with no *password*, then any current user password is removed.

OWNER="*password*" sets the owner password to *password*.If this parameter is omitted, the owner password remains the same. If OWNER= is specified with no *password*, then any current owner password is removed.

PROT=*level* specifies the protection level that is enforced if the user password is specified. If this parameter is omitted, the level is unchanged. You have to give a file an OWNER password before you can set PROT. The optional levels for access to a file are:

EXEC	Execute only
READ	Read and execute
UPDATE	Update (change existing records), read, and execute
WRITE	Write, update, read, and execute
RENAME	Rename, write, update, read, and execute
REMOVE	Remove, rename, write, update, read, and execute (allows total access except for changing attributes with the ATTRIB command)
FULL	Allows total access

VIS specifies the filespec as visible in the directory

INV specifies the filespec as invisible in the directory. Use the INV parameter to reduce the number of files that TRSDOS displays when you issue a DIR command.

You can abbreviate the levels of PROTection to their first two letters except for RENAME and REMOVE, which you can abbreviate to RN and RM respectively.

For disk ATTRIBs, the parameters are:

LOCK protects all visible files not currently protected by setting their user and owner passwords to the disk master password.
UNLOCK removes the user and owner passwords from visible files if their passwords match the disk master password.
MPW="*password*" states the disk's current master password. If you don't specify this option, TRSDOS prompts you for it, if the password is not PASSWORD.
NAME=["*disk name*"] specifies the new disk name. If this parameter is omitted, the disk name remains the same.
PW=["*password*"] sets the new disk master password to *password*. If this parameter is omitted, the disk master password remains the same.

PW cannot be abbreviated.

drive defaults to Drive 0.

Assigning Protection Attributes To a File

Using the Owner and User Passwords. Passwords are first assigned when the file is created. At that time, the **owner** and **user passwords** are set at the same value (either the password you specified, or a blank password if you did not specify one).

ATTRIB allows you to assign a file two different passwords. The user password could be for the operator. It protects a file's contents at a certain protection level (set by PROT). For example, if you want an operator to have limited access to a file, you can set the PROTection level to READ. Then, using the user password, the operator will be able only to read (list) and execute the file, not change, rename, re-attrib, or remove it.

In the same manner, the owner password could be for the programmer. Using the owner password, the programmer could change, remove, re-attrib, or rename the same file. (When you use the owner password to access a file, TRSDOS ignores the PROTection level.)

In short, the user password allows limited access to a file and the owner password allows total access.

Examples

```
ATTRIB CUSTFILE/DAT:1
(USER=,OWNER="BOSSMAN",PROT=READ) (ENTER)
```

sets the user password blank (so no password is necessary to access the file), sets the owner password to BOSSMAN, and sets the protection level to read and execute.

```
ATTRIB CUSTFILE/DAT.BOSSMAN
(USER="SECRET",PROT=EXEC,INV) (ENTER)
```

re-attribs CUSTFILE/DAT. Note that the owner password BOSSMAN was required to re-attrib the file. Now, CUSTFILE/DAT has the user password SECRET, keeps owner password BOSSMAN, has the protection level of execute only, and is invisible in the directory.

Assigning Protection Attributes To a Disk

The ATTRIB command also allows you to change the disk name, the disk master password, and the password protection of all visible filespecs.

Examples

```
ATTRIB (UNLOCK,NAME="MYDISK") (ENTER)
```

removes all user and owner passwords from the visible filespecs on Drive 0 if the filespecs' current password matches the disk master password. It also changes the disk name to MYDISK. Since the current master password was not specified with the MPW parameter, your computer asks you for it (if it is other than PASSWORD) before it executes this command.

```
ATTRIB :1 (NAME="DATA",PW="SECRET",MPW="BOSSMAN")
(ENTER)
```

sets the disk name in Drive 1 to DATA, changes the master password to SECRET if the current disk master password is BOSSMAN.

```
ATTRIB (LOCK) (ENTER)
```

prompts you for the disk's master password (if other than PASSWORD) and changes the user and owner passwords of all visible, non-password protected files to the disk's current master password. Since no drive was specified, the command is carried out on Drive 0.

```
ATTRIB :1 (NAME) (ENTER)
```

prompts you for Drive 1's disk master password (unless it is PASSWORD). It then prompts you for the new disk name.

Error Conditions

If you specify invalid values or omit necessary quotes for an ATTRIB parameter, TRSDOS displays the message "Attribute specification error." Check the parameters and try the command again.

If the disk master password is not PASSWORD, you must specify it when executing an ATTRIB command from a JCL file. The JCL file cannot prompt you for the password during execution. If you omit the password, TRSDOS displays an "Invalid master password" error message.

Sample Uses

Suppose you have a data file, PAYROLL, and you want an employee to use the file in preparing paychecks. You want the employee to be able to read the file but not to change it. Then use a command like:

```
ATTRIB PAYROLL  (USER="PAYDAY", OWNER="BANANA",
PROT=READ) (ENTER)
```

Now tell the clerk to use the password PAYDAY (which allows read only); while only you know the password, BANANA, which grants total access to the file.

AUTO

AUTO [***parameters***] [*][***command line***]	**Command**

Stores an AUTO *command line*. This *command line* automatically executes whenever you start up or reset TRSDOS. (That is, after you enter the date and time, TRSDOS loads, executes the command line, and displays the TRSDOS Ready or BASIC prompt.)

You can use AUTO to automatically run a program after you type in the date.

command line is limited to 74 characters in length.

The *parameters* are:

[:*drive*] specifies on which drive to store the AUTO command line.
?[:*drive*] displays any AUTO command line stored on *drive*.
=[:*drive*] executes any AUTO command line stored on *drive*.

In all cases, the default drive is Drive 0.

In most cases, you can override the AUTO command during start-up or reset by (1) holding down the (ENTER) key, or (2) pressing (BREAK) while the auto command is executing.

The exception to this is when you store the AUTO command with the * parameter (which disables the (BREAK) key and the ability of the (ENTER) key to override AUTO).

If the AUTO command disables the (BREAK) key and the program is non-functional, gaining control of the disk requires several steps. To regain control:

1. Start up the system with another non-AUTOed disk in Drive 0.

   Note: The two disks must contain the same version of TRSDOS.

2. When `TRSDOS Ready` appears, place the non-functional disk in Drive 0.
3. Type AUTO and press (ENTER), and the runaway AUTO command is removed from the disk.

Use the :*drive* parameter to place an AUTO command on a drive other than Drive 0.

Examples

`AUTO BASIC` (ENTER)

loads the BASIC program whenever you start up or reset on Drive 0.

`AUTO` (ENTER)

Turns off the AUTO function currently stored on Drive 0.

`AUTO *DO INIT/JCL:1` (ENTER)

executes the DO file on Drive 1 named INIT/JCL whenever you start up or reset. Notice that the * parameter is used. This means the operator cannot use (ENTER) to halt the auto command; (BREAK) is also disabled.

```
AUTO :1 DEVICE (ENTER)
```

places the AUTO command DEVICE on Drive 1.

```
AUTO ?:1 (ENTER)
```

displays the AUTO command on Drive 1.

```
AUTO =:1 (ENTER)
```

executes the AUTO command on Drive 1.

Error Conditions

To place an AUTO command on a disk, it must be write-enabled.

The system does not check the command line for errors when you first enter the AUTO command line. Errors are detected when the system starts up or when the AUTO= command is executed.

Sample Use

Suppose you want the DEVICE command to execute automatically when you restart your computer.

Do this by issuing the command:

```
AUTO DEVICE (ENTER)
```

BACKUP

	Utility
BACKUP [***partspec* ¦ *-partspec***]**:*source drive***[TO] **:*destination drive***[(***parameters***)]	

Duplicates (backs up) all or some of the files from *source drive* to *destination drive*.

You can use BACKUP to copy the contents of one disk to another.

If you specify *partspec*, BACKUP duplicates all files that match *partspec*. If you specify *partspec*, preceded by a hyphen (-), BACKUP duplicates all files that do not match *partspec*. If you include a drive number with *partspec*, you must include the colon(:).

If you omit *parameters* and *partspec*, TRSDOS performs a "mirror-image backup" and duplicates all files. If the disk types are different, TRSDOS performs a "backup reconstruct" and duplicates only visible files.

Note: In most cases, you can see which files a BACKUP command would duplicate by issuing a DIR command of *source drive* using the same *partspec* and *parameters* as the BACKUP command.

If you do not specify *source drive* and *destination drive*, the system prompts you for them. If the source disk has a Master Password other than PASSWORD, and you do not state it with the MPW= parameter, the system prompts for it as well.

If the destination drive is not ready, the message "Insert DESTINATION disk <ENTER>" is displayed. Insert the destination disk and press (ENTER) to continue, or press (BREAK) to return to TRSDOS Ready.

The destination disk must be formatted before the backup begins. To format a disk, see the FORMAT command.

The parameters are:

- MPW="*password*" specifies the source disk's Master Password
- SYS backs up system files as well as the visible files
- INV backs up invisible files as well as the visible files
- MOD backs up files that have been modified since the last backup
- QUERY=YES questions you about each file before it is backed up
- OLD backs up only those files that already exist on the destination disk
- NEW backs up only those files that do not already exist on the destination disk
- X allows backups with no system disk in Drive 0

DATE="*date*" backs up files modified on *date*.
="*date1-date2*" backs up files modified on or after *date1* and on or before *date2*.
="*date-*" backs up files modified on or after *date*.
="*-date*" backs up files modified on or before *date*.

Dates must be in the format *mm/dd/yy*.

MPW cannot be abbreviated.

When you specify QUERY=YES, the system questions you for each file before it is copied. Answer by pressing:

(Y)	to copy the file.
(N) or (ENTER)	to bypass the file and move on to the next one.
(C)	to copy the file, turn off the Query function, and automatically copy all remaining files.

After you type the BACKUP command, TRSDOS automatically performs one of the three types of backups: "mirror image," "backup by class," or "backup reconstruct." The difference between the three is of technical interest and is discussed in "General Information."

NOTE: A backup by class and backup reconstruct require two disk drives.

For information on backing up "backup limited" diskettes, see Appendix M, "Backup Limited Diskettes."

Backups With the (X) Parameter

When you specify the (X) parameter, you do not have to have a system disk in Drive 0 when you back up a disk. TRSDOS prompts you to insert the proper disks in the proper drive.

Examples

```
BACKUP $:0 :1 (SYS,INV) (ENTER)
```

examines all files on the disk in Drive 0 and copies all files to Drive 1, because all files match the $ partspec. The partspec causes a backup by class.

NOTE: You can use this command to force a backup by class in situations where a mirror image would normally be performed. For example, it reduces fragmentation of files on the source disk by copying them in a more contiguous manner onto a newly formatted destination disk.

```
BACKUP :0 :1 (MOD,QUERY=YES,MPW="SECRET") (ENTER)
```

copies all visible files from Drive 0 to Drive 1 that have been modified (written to) since the last backup. It questions you for each file before it is copied, showing the file's mod date and flag. The (MPW=) parameter states the Master Password, so the system does not prompt you for it.

```
BACKUP $/CMD:0 :1 (ENTER)
```

copies all visible files with the extension /CMD from Drive 0 to Drive 1. If the file already exists on Drive 1 it is overwritten. No other files on Drive 1 are touched. A backup by class is performed.

```
BACKUP /$$S:1 :2 (ENTER)
```

backs up all files whose extensions are three characters long and end with the letter S. The $ wildcard masks the first two characters of the extension, so extensions such as /BAS, /TSS, and /TRS form a match. A backup by class is performed.

```
BACKUP :1 :1 (ENTER)
```

backs up between two disks in Drive 1. You are prompted to switch the source disk and destination disk at the appropriate times.

The disks used in this type of backup must allow a mirror image backup, or the backup aborts.

This command and the following command could be used to back up a data disk.

```
BACKUP :0 :1 (X) (ENTER)
```

backs up the disk in Drive 0 to the disk in Drive 1. Its main use is to back up non-system disks, such as data disks, in a two-drive system.

When you use this parameter, you are prompted to insert the proper disk in Drive 0. You may be prompted to re-insert a system disk into Drive 0 during certain backups.

When the backup is complete, you are prompted to insert a system disk back in Drive 0.

```
BACKUP -/CMD:0 :1 (ENTER)
```

backs up all visible files from Drive 0 to Drive 1, EXCEPT those files that have a /CMD extension. A backup by class is performed.

```
BACKUP :1 :2 (NEW,QUERY=YES) (ENTER)
```

backs up only those visible files from Drive 1 that do not already exist on Drive 2. You are prompted before each file is moved.

```
BACKUP /ASM:3 :2 (DATE="05/06/82-05/10/82")
(ENTER)
```

backs up all visible files with the extension /ASM, whose modify dates fall on or between the specified dates.

Error Conditions

The destination disk must be formatted before the backup begins. To format a disk, see the FORMAT command.

For a backup by class, if the backup is to include system files, the destination disk must be newly formatted. BACKUP can't create a system disk if the destination disk contains data files where system files would reside. (Existing files may be using certain areas needed by the system.)

If you are backing up the entire disk, TRSDOS compares the source and destination disk Disk ID's to make sure they are identical. If the master passwords or disk names differ, you see the following message:

```
Destination disk ID is different --
NAME=disk name
     DATE=mm/dd/yy
Are you sure you want to backup to it <Y,N>?
```

Press (N) to abort the BACKUP or (Y) to continue. Press (ENTER).

If the disks' master passwords differ, the following message appears:

```
Destination disk ID is different -- NAME=disk
name
     DATE=mm/dd/yy
Enter its Master Password or <Break> to abort:
```

Press (BREAK) to abort the BACKUP or enter the password to continue.

If the source and destination disks have a different number of cylinders, the following message appears:

```
Cylinder counts differ - Attempt mirror-image
backup ?
```

Answer this question with (Y) to attempt a mirror image backup or with (N) to force a backup reconstruct. Press (ENTER).

If a mirror image backup is not possible, you get the error:

```
Backup aborted, destination not mirror-image
```

This appears if the destination disk is missing a cylinder that contains information on the source disk. This might be the case if the destination disk is formatted with fewer cylinders than the source disk, or if cylinders are locked out on the destination disk when it is formatted. You can use the FREE library command to check the destination disk for locked out cylinders.

After all the cylinders that contain data are copied to the destination disk, BACKUP attempts to remove the modification flags from the files on the source disk. If the disk is write protected, the following warning message appears:

```
Source disk is write protected; MOD flags not
updated
```

Backup by class may NOT be done on a single drive.

Mirror Image Backup. A mirror image backup is basically a cylinder-for-cylinder copy from the source to the destination disk. (Only those cylinders that actually contain data are copied.) When the backup is complete, the destination disk is an exact copy, or mirror image, of the source disk.

Backup By Class. A backup by class takes place if you specify a partspec or any parameter except X or MPW in the command line.

Backup Reconstruct. A backup by class and a backup reconstruct function identically. The only difference is that while **you** initiate a backup by class, the **system** initiates a backup reconstruct.

On certain TRSDOS application programs, you can only make a limited number of backups. And, when you make a backup on one of these programs, the source disk has to be write-enabled during the backup or the backup fails. See Appendix M/Backup Limited Diskettes.

Backups With the (X) Parameter. This parameter allows you to back up data disks of different sizes or capacities on a two-drive system (using backup reconstruct). When you use the (X) parameter to backup non-system disks of different sizes or capacities, system modules 2, 3 and 10 must first be put into memory with the SYSTEM (SYSRES = *number*) command. Remember that the (X) parameter is used only when there is a non-system disk in Drive 0. When you specify X, BACKUP prompts you to insert the data disk.

Mirror Image Backup. TRSDOS makes a mirror image backup if the source and destination disks' size and density are identical, and if you specify no partspec or parameters (except X or MPW) in the command line. The number of cylinders doesn't need to be identical as long as the destination disk has at least as many cylinders as the source disk.

The date on the destination disk shown with the DIR or FREE library commands is changed to the current system date.

After the backup, the destination disk has its directory on the same track as the source disk regardless of where it was before the backup. The information on the destination disk is updated to reflect its true cylinder count and available free space.

Backup By Class. This type of backup does a file-for-file copy from the source to the destination disk. Files that are fragmented (spread over more than one extent) on the source disk are consolidated (if possible) on the destination disk.

Unlike a mirror image backup, files that exist on the destination disk but are not on the source disk are not touched in the backup. When

the backup is complete, the destination disk contains all files moved from the source disk plus any other files that existed on the destination disk before the backup began.

The destination Disk ID is not changed by the backup.

When the file SYS0/SYS is included in a backup by class, the destination disk is configured in the following manner:

1. The state of the SYSGEN (on or off) is changed to match that of the source disk.
2. The initial date and time prompts (on or off) on power-up are set to match those of the source disk.
3. The default drive configurations match those of the source disk.

Backup Reconstruct. The system performs a backup reconstruct when the size or the density differs between the source and destination disks.

DIR/SYS and BOOT/SYS are not moved to the destination disk in this type of backup.

If you want all of the files to be moved, then you must use the command BACKUP :*source drive* :*destination drive* (SYS,INV). This moves visible, invisible, and system files to the destination disk.

When you are performing a backup by class or backup reconstruct, TRSDOS may display a Disk is full — Insert new formatted destination disk <ENTER> warning message. This is not an error message. TRSDOS is warning you that the destination disk is full and all of the files on the source disk are not on the destination disk. Remove the destination disk and insert a new formatted disk. Press **(ENTER)**.

TRSDOS is most likely to display this message when you are backing up a hard disk or when the destination disk is partially full before the backup. If a portion of a file is on the destination disk when TRSDOS realizes that the disk is full, TRSDOS removes that portion of the file. A file is never divided between two diskettes. TRSDOS also indicates which files are on which destination disk.

Hard disk users should note that system files are stored in specific places in the directory. If you use BACKUP to move the visible files and then repeat the command with the SYS option, the backup could be cancelled if the directory positions required for the system files are already in use. If this happens, you can use the PURGE command to delete the files that were moved to the disk, and then give the BACKUP command with the SYS option.

Sample Use

Suppose you have a payroll disk where all of the new employees have a file with an extension of /NEW and all of the old employees have a file with an extension of /OLD.

Now suppose you want to have two separate disks: one with old employee files and one with new employee files. You could issue the command:

```
BACKUP /NEW:0 :1 (ENTER)
```

to move all of the files of new employees from the master disk in Drive 0 to another disk in Drive 1.

BOOT

BOOT [*keys*]	**Command**

Resets (boots) TRSDOS by returning it to its original start-up condition.

You can use BOOT to return your computer to the TRSDOS copyright and startup message.

The keys are:

- (CLEAR) allows no sysgened configuration to take place.
- (ENTER) allows no breakable AUTO commands to occur.
- (D) enters the system debug. No sysgened configuration is loaded.

Note: When you use one or more of these keys, you must press and hold them down when the screen is erased and keep them down until the TRSDOS Ready message or the DEBUG display appears. If you are prompted for the date, you must hold down the keys as soon as you type the date and press (ENTER). If you don't press the keys in time, simply reset and hold the keys down as soon as the screen clears.

If the diskette in Drive 0 contains a CONFIG/SYS file, TRSDOS displays the message ** SYSGEN ** in the lower left corner of the display while the CONFIG/SYS file is loading.

Note: CONFIG/SYS files that were created using previous versions of TRSDOS, Version 6.0 or 6.1, cannot be used with TRSDOS Version 6.2. You cannot copy a configuration file to a disk. You must use SYSGEN to create a new CONFIG/SYS file for Version 6.2.

BOOT loads the TRSDOS system in floppy Drive 0 back into the computer. It returns the computer back to its normal power-up configuration as if the system had been turned off and then turned on again.

Examples

Remember to hold down the *key* after you press (ENTER) until you see TRSDOS Ready or the debug display.

```
BOOT (ENTER)
```

resets the system.

```
BOOT (ENTER) (CLEAR)
```

returns the system to its original start-up condition and ignores any sysgened configuration.

```
BOOT (ENTER) (ENTER)
```

returns the system to its original start-up condition and ignores any breakable AUTO commands.

BOOT (ENTER) (D)

returns the system to its original start-up condition and enters the system debug. No sysgened configuration is loaded, and any AUTOed command is not executed. Note: If the AUTOed command is unbreakable, (D) is ignored.

BUILD

BUILD *filespec* [(*parameters*)]	**Command**

Lets you enter data (such as commands) and save it on disk as *filespec*.

If you omit an extension to *filespec*, TRSDOS assumes the extension /JCL. If you include only the slash for extension, TRSDOS does not assume an extension.

You can use BUILD to make a file on a disk.

The parameters are:

> HEX accepts data in hexadecimal format only.
> APPEND appends the BUILD data to the end of *filespec*.

Although you can build any type of data file with this command, it is mainly for creating files to be executed with the DO command, KSM/FLT, or the PATCH utility.

The HEX parameter lets you input data in hexadecimal form (see Appendix C for a listing of hexadecimal characters). You can use hex to generate control characters and graphics symbols which are not available from the keyboard.

The APPEND parameter lets you add data to the end of an existing file.

Some programs (such as SCRIPSIT) place their own marker at the end of a file. If this marker is in the file, you cannot append BUILD data to it unless you:

- Use the BUILD command to create a new file containing the information you wish to append.
- Use the APPEND library command with the STRIP parameter to properly append the new information to the existing file.

Building a File

When you enter the BUILD command with a non-existing *filespec*, BUILD creates the file and then allows you to insert lines.

You can enter a command line of up to 255 characters. JCL files are limited to 79 characters per line. To end a line, press (ENTER).

To end the file, press (CTRL)(SHIFT)(@) at the beginning of a new line. The system returns you to TRSDOS Ready.

Examples

```
BUILD DISPLAY:2 (ENTER)
```

creates a new file named DISPLAY/JCL on Drive 2. TRSDOS allows you to insert lines. Type:

```
DEVICE (ENTER)
FREE :0 (ENTER)
FREE (ENTER)
(CTRL)(SHIFT)(@)
```

The first three lines insert the DEVICE, and FREE :0 and FREE commands into the "DISPLAY" file. Pressing (CTRL)(SHIFT)(@) tells TRSDOS that you are finished entering command lines. The system returns to TRSDOS Ready.

Now, whenever you type:

```
DO DISPLAY (ENTER)
```

TRSDOS executes the file by displaying the device table, the free space map of Drive 0, and the free space information for all enabled drives.

```
BUILD MYKEYS/KSM (ENTER)
```

builds MYKEYS on the first available drive. Since the /KSM extension was used, a KSM file is built. See the KSM/FLT filter in Appendix I for more information.

```
BUILD SPECIAL/:0 (ENTER)
```

builds SPECIAL on Drive 0. Adding the "/" allows SPECIAL to be built without an extension.

```
BUILD MYJOBS/JCL (APPEND) (ENTER)
```

searches all available drives for MYJOBS/JCL (until it is found) and adds the information from this build to the end of the file. If MYJOBS/JCL is not found, the file is built on the first available drive.

```
BUILD MYPROGA/FIX:0 (ENTER)
```

builds MYPROGA/FIX, which is to be used with PATCH. See the PATCH utility for more information.

```
BUILD DISPLAY/BLD (HEX) (ENTER)
```

builds a file on the first available drive, allowing data to be entered in hexadecimal format. Information is entered into this file as hexadecimal bytes (with no spaces or other delimiters between them).

The HEX parameter allows you to enter characters not directly available from the keyboard, such as control, printer control, and graphics characters. You can enter any one-byte character value.

A hex build allows 127 hex byte representations (254 characters) per logical line. Logical lines may continue on more than one physical line as long as a "0D" logical line terminator does not appear. Also, more than one logical line can appear on one physical line.

To create a character string containing graphics characters, type:

```
818A90A10D (ENTER)
```

This line contains the hexadecimal bytes 81, 8A, 90, and A1. Notice that the byte values are packed together. "0D" ends a logical line, and (ENTER) ends a physical line.

If a non-hex digit is entered, the error message "Bad hex digit encountered" is displayed and the build aborts.

Error Conditions

If you omit the APPEND parameter and specify a filespec that already exists, TRSDOS displays a "File already exists" error message.

Sample Use

Suppose you want to build a file to be used with the PATCH command. Issue the command:

```
BUILD PROG/FIX (ENTER)
```

and enter the patch lines.

Command

CAT [***partspec*** **¦** ***-partspec***][**:**][***drive1***][**-**][**:**][***drive2***][**(*parameters*)**]

Displays the directory for one or more drives.

The CAT command displays the names of the files on a drive or a range of drives.

If you specify *partspec*, CAT displays only filenames that match *partspec*. If you specify *partspec*, preceded by a hyphen (-), CAT displays all filenames that do not match *partspec*. If you include a drive number with *partspec*, you must include the colon.

Colons are optional in the syntax of the CAT command except when:

- You specify partspec with a drive number.
- You include a colon for *drive1*, you must omit the colon for *drive2*.

If you omit the drive numbers, CAT displays the filenames on all enabled drives. You can include the hyphen to specify a range of drive numbers. If you specify:

drive1-drive2	displays the directory for *drive1* through *drive2*.
drive1-	displays the directory for all drive numbers equal to or greater than *drive1*.
-drive2	displays the directory for Drive 0 through *drive2*.

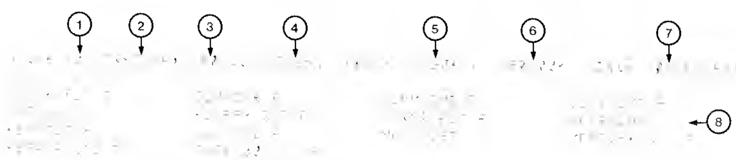
Specifying parameters allows you to select which filenames CAT prints on the screen or line printer. You must enclose parameters in parentheses. If you include more than one parameter, separate each with a comma. You cannot abbreviate the SORT and SYS parameters. The parameters are:

ALL	displays all directory information for the specified diskette(s). If the directory information is longer than 24 lines and you do not specify the NON parameter, CAT displays 24 lines of information and waits for you to press a key before displaying the next 24 lines.
INV	displays all filenames, visible and invisible.
MOD	displays filenames modified since the last backup.
NON	enables non-stop display mode. When the directory information fills a screen, NON scrolls lines off the top of the screen.
PRT	the directory display prints on the printer and the screen. If you include PRT, CAT assumes the NON parameter.
SYS	displays system and visible filenames.

DATE =“*date1-date2*” displays the filenames that have been modified on or after *date1* and before or on *date2*.
=“*date*” displays the filenames that were modified on *date*.
=“*date-*” displays the filenames modified on or after *date*.
=“*-date*” displays the filenames that were modified on or before *date*.

Dates must be in the format *mm/dd/yy*.

SORT=NO does not sort the filenames. CAT assumes SORT=YES, except when you are in an application program, including BASIC. Sorting is by alphabetical order.



1. Drive Number.
2. Disk Name.
3. Number of cylinders on the disk.
4. Density of the disk.
   DDEN = double density
   SDEN = single density
   Hard = hard disk
5. Amount of free (unused) space on the disk.
6. Amount of space (used and unused) on the disk.
7. Creation date (the date the disk was formatted or was the destination in a mirror-image backup).
8. Names of the files on the disk. The filenames are sorted alphabetically unless you specify SORT = NO.

For a detailed explanation of each line with the ALL option, see the DIR command.

Examples

```
CAT (ENTER)
```

displays the filenames of all visible files on all enabled drives.

```
CAT :1- (ENTER)
```

displays the visible filenames on enabled drives that are equal to or greater than one.

```
CAT (INV,SYS) (ENTER)
```

displays all filenames (visible, invisible, and system) on all enabled drives.

```
CAT (ALL) (ENTER)
```

displays the ALL directory information for enabled drives. If the information is longer than 24 lines, CAT displays 24 lines of the directory and waits for you to press a key before displaying the next 24 lines. See the DIR command for a complete description of a directory with the ALL parameter.

```
CAT 0 (PRT,MOD) (ENTER)
```

prints and displays the visible filenames on Drive 0 that have been modified since the last backup. When you specify the PRT parameter, CAT assumes the NON parameter and displays the filenames on the screen without pausing. You can press (SHIFT) and (@) to pause the display. Press any key to continue.

```
CAT 1 (DATE="04/16/84-")
```

displays the visible filenames on Drive 1 modified on or after April 16, 1984.

```
CAT -1 (DATE="-07/20/84") (ENTER)
```

displays the visible filenames on Drives 0 and 1 modified before July 20, 1984.

```
CAT /CMD:0 (ENTER)
```

displays the visible filenames on Drive 0 that have the extension /CMD.

```
CAT -/CMD:0 (ENTER)
```

displays the visible filenames on Drive 0 that do not have the extension /CMD.

Error Conditions

If you specify a drive number that does not exist or that is not enabled, CAT displays an "Illegal drive number" error message.

If you specify a range of drives and include a colon with each, CAT assumes that the second colon is a drive number and displays an "Illegal drive number" error message. Try the command again and only include the colon for the first drive number.

If you specify an enabled drive number that does not contain a formatted disk, CAT displays a "[No Disk]" message.

CLS

CLS	Command

Clears the screen, positions the cursor in the upper left-hand corner of the display, and sets the display to 80-column mode.

In some application programs, and at TRSDOS Ready, you can also clear the screen by pressing (SHIFT) and (CLEAR) at the same time. In some applications, such as JCL, you cannot press keys. That is when CLS is particularly useful.

COMM

COMM *devspec* [(*parameters*)]	**Utility**

Lets two computers communicate via a device, usually the RS-232-C communications line.

You can use COMM to let your computer talk with another computer.

COMM lets your computer:

- be used as a terminal in communicating with another computer.
- transfer files to and from another computer.
- spool output from the other computer to your printer.

Using COMM, you can access:

- Bulletin Board Systems
- News and Information Systems
- Timesharing Systems
- Electronic Mail Services

COMM can also communicate with systems that support XON/XOFF (Proceed/Pause) protocol. This is a protocol that uses two control codes named Device Control 1 and Device Control 3. (The Device Control codes are discussed in the (CLEAR) (SHIFT) (*) command section.)

devspec is usually *CL, the RS-232-C communications line.

Note: Before you can use *CL, you have to SET it to its driver program COM/DVR. See Appendix I.

The *parameters* are:

XLATES=*X'aabb'* translates a character being sent.
XLATER=*X'aabb'* translates a character being received.
XON=*X'cc'* changes the XON code.
XOFF=*X'cc'* changes the XOFF code.

aa is the character to be translated.
bb is the character *aa* is translated into.
cc is the new value of XON or XOFF.

Enter hexadecimal values in the format X'nnnn'.

NULL=OFF	prevents any nulls (ASCII value 0) from being received.

XLATES and XLATER can be abbreviated to XS and XR.

XLATES and XLATER let you translate a character that you send and a character that you receive from another computer. (Only one character can be translated in each direction at any one time.)

Error Conditions

If there is not enough memory available to establish the device buffers for COMM. TRSDOS displays an "Insufficient memory to establish buffers" error message. Remove some modules from memory that you are not using or press RESET to release memory.

If you omit *devspec* or specify a device other than the device specified with SET statement, TRSDOS displays a "Comm Line driver not specified" error message.

Example

Suppose you are using COMM as a terminal to communicate with another computer, and you want to print a right bracket ( ]). It appears that you can't because there is no key on your keyboard that produces this character.

Use the XLATES parameter to produce a ( ]) by entering another character from the keyboard. Type:

```
XLATES=X'025D'
```

Now when you press CTRL B (hexadecimal 02), your computer sends the code for a right bracket (hexadecimal 5D) to the other computer. Since your computer can display a right bracket when it receives a hex 5D, there is no need to use XLATER to translate the received character

Characters that you receive from another computer can be translated to a different symbol using XLATER. Use the same method that we used for XLATES.

See Appendix C for a list of characters and their hex values.

The Function Keys

The Function Keys are used to:

1. Direct the flow of data (text or software) from device to device.
2. Enable and disable certain functions of COMM, including XON/XOFF and full/half duplex operation.

The Function Keys are divided into two groups: (1) the Application keys and (2) the Action keys.

The Application and Action keys are achieved by holding down all of the keys in the sequence, such as CLEAR 6 (hold down CLEAR and then press 6).

The Application Keys

The Function Keys CLEAR 1 through CLEAR 6 designate what device an action applies to.

Function Key	Device	Abbreviation
CLEAR 1	Keyboard Device	(*KI)
CLEAR 2	Display Device	(*DO)
CLEAR 3	Printer Device	(*PR)
CLEAR 4	Communications Line Device	(*CL)
CLEAR 5	"Data Send" Device	(*FS)
CLEAR 6	"Data Receive" Device	(*FR)

Action Keys

The remaining Function Keys perform an action. Some action keys require you to specify an application key (CLEAR 1 through CLEAR 6) before you can perform the action.

CLEAR 7

Causes the contents of the "Data Received" (*FR) area in memory to be written to disk. This is called "Dump-To-Disk" or DTD. DTD may be turned ON before or after a file is received. (You must turn DTD ON if a file will exceed the size of the *FR memory area.)

When you start COMM, DTD is ON. When you perform an *FR RESET (CLEAR 6 CLEAR 0), DTD is turned OFF. To turn it ON again, press CLEAR 7 followed by CLEAR ;

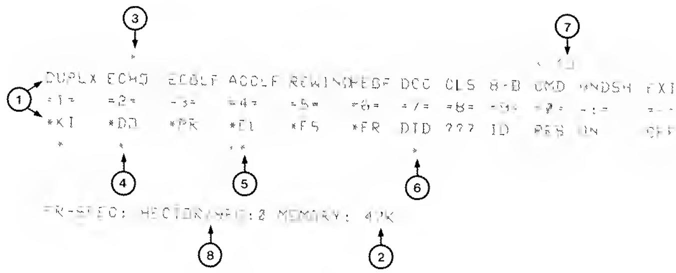
If you are writing data to floppy disks and the RS-232 port is running at a speed higher than 300 baud, you have to wait until you receive an entire file before turning DTD ON.

CLEAR 8

Displays the MENU of Function Keys on the display. You can use this command any time.

The display goes from left to right. This is not intended to be a complete menu, but a built-in "quick reference" card.

The screen display is altered to display the menu. Any data you receive while the menu is displayed is not lost because COMM saves the data in a special area of memory. This data appears on the screen after the menu is displayed.



1. The devices and functions. (The asterisks above and below the function keys indicate that the function is active.)
2. The amount of available memory.
3. Asterisks for the shifted function keys.
4. Asterisks for the unshifted function keys.
5. Two asterisks denote a device capable of both input and output.
6. One asterisk denotes a device capable of either input or output.
7. If HANDSHAKE is active, the auto XOFF character selected is shown in hex.
8. Displayed filespec of any *FS or *FR filespec.

(CLEAR)(9)

Specifies what file to use when you send or receive data. After you specify the file, your computer opens it. If you specify a file that does not exist, COMM creates it.

To specify the name of a receiving file, press (CLEAR)(6) followed by (CLEAR)(9) and answer the following prompt:

```
File Name,
```

COMM opens the file but does not set aside an area of memory to receive the data, so any incoming data is ignored.

To save incoming data, enter the command (CLEAR)(6) followed by (CLEAR)(:). Now, data received is placed in the "Data Received" (*FR) area of memory, and the data is eventually placed in the file you specified. (See (CLEAR)(:) for more information on activating devices.)

If a file is already open, the system aborts your (CLEAR)(9) command and prints the warning message:

```
File Already Open
```

This warning prevents you from opening another file before closing this one. This protection also applies to files associated with the "Data Send" (*FS) area of memory.

[CLEAR][0]

Closes either a receive file or a send file. You must close a receive file so its directory can be updated, and so you can receive another file. If you reset a device, its buffer is cleared.

You must turn OFF the *FR or *FS device before you can close the associated file. (See [CLEAR][–] for information on turning OFF devices.)

[CLEAR][:]

Turns ON a device. This is the second command of a two-command sequence.

For example, if you want to turn ON the printer, first press [CLEAR][3] to indicate that you want to do something with the printer, and then press [CLEAR][:] to indicate that you want to turn it ON. Press [CLEAR][3] followed by [CLEAR][–] to turn the printer OFF.

[CLEAR][–]

Turns OFF a device. This is the second command of a two-command sequence.

For example, if you want to turn the printer OFF, first press [CLEAR][3] to indicate that you want to do something with the printer, and then press [CLEAR][–] to indicate that you want to turn it OFF. Press [CLEAR][3] followed by [CLEAR][:] to turn the printer ON.

[CLEAR][SHIFT][!]

This is the DUPLEX control, which allows you to select Full-Duplex or Half-Duplex.

Full-Duplex and Half-Duplex indicate how data is sent from one computer to another on an RS-232-C line.

- Half-Duplex is used with a computer that cannot read data while it is sending it or send data while it is receiving it. A [CLEAR][SHIFT][!] followed by a [CLEAR][:] indicates Half-Duplex.

- Full-Duplex is used with a computer that can read data while it is sending it or send data while it is receiving it. A [CLEAR][SHIFT][!] followed by a [CLEAR][–] indicates Full-Duplex operation.

When you start COMM, it is set to Full-Duplex (Duplex off).

Some computers and terminals combine Duplex and Echo functions, so you should know something about the computer or terminal you are communicating with.

(CLEAR)(SHIFT)(")

Controls character "Echo"ing. Press (CLEAR)(SHIFT)(") followed by (CLEAR)(;) to turn Echo ON.

You should turn Echo ON if you are communicating with a computer or terminal that is operating in full-duplex, but does not display a copy of characters it is transmitting to you (called Local Copy).

Turning Echo ON causes your computer to transmit each character it receives back to the computer that sent it. This lets the person operating the other computer see what is being transmitted.

Caution: If both ends are set for Echo ON, then the first character sent is echoed back and forth indefinitely — or until one end turns Echo OFF.

(CLEAR)(SHIFT)(#)

This command controls Echoing linefeeds. When enabled, any carriage return your computer receives causes a linefeed character to be transmitted back to the other computer.

This command is useful since there are a large number of terminals and computers that treat a carriage return (ASCII 13) and linefeed (ASCII 10) as separate functions.

When you are communicating with another TRS-80 computer, you can turn OFF this function by pressing (CLEAR)(SHIFT)(#) followed by (CLEAR)(-).

(CLEAR)(SHIFT)($)

Controls the ability of your computer to accept a linefeed. COMM usually ignores the first linefeed after a carriage return, since most computers send both a carriage return and a linefeed.

In most cases, this command is not necessary on TRS-80's where an (ENTER) is treated as both a carriage return and linefeed.

(CLEAR)(SHIFT)(%)

Positions to the start of an *FR or *FS file, so you can start again. For example, if you are receiving a file and it aborts with an error, you can start over by pressing (CLEAR)(6) followed by (CLEAR)(SHIFT)(%). Then you can attempt to receive the file again.

(CLEAR)(SHIFT)(&)

Appends new data to the end of a file. This command applies to the "Data Received" (*FR) area of memory only. If you open an existing file and then press (CLEAR)(6) followed by (CLEAR)(SHIFT)(&), you can append new data to the end of the file.

(CLEAR)(SHIFT)(,)

Displays control characters that are being received or sent. You can use this command to detect if you are receiving unwanted control characters. If you are receiving unwanted control characters, you can use the XLATER to translate them.

(CLEAR)(SHIFT)(()

Erases the contents of the screen and places the cursor in the upper left corner. No data is transmitted.

(CLEAR)(SHIFT)())

When followed by an ON command ((CLEAR)(:)), your computer uses all 8 bits of a character it receives. Normally, bit 8 is either not present or invalid, so COMM removes it from each character it receives.

Do not turn this option ON unless the RS-232-C word length is set to 8. You can use the SETCOM library command to set the word length before you enter COMM.

(CLEAR)(SHIFT)(0)

Allows you to enter a TRSDOS **library command** from COMM.

For example, when you type:

```
DEVICE (ENTER)
```

the device table is displayed on your screen. The message "Command complete" is displayed below the device table.

NOTE: If the specified library command attempts to change HIGH$, the command aborts and TRSDOS returns you to COMM.

(CLEAR)(SHIFT)(*)

This command controls the handshaking on the data line.

Handshaking is the agreed-upon method that two communicating computers use to control the flow of data between them. If this option is turned ON, COMM responds to the following codes when your computer receives them from the communications line:

Symbol	Value			Description
DC1	17	X'11'	—	Resume transmission (XON or Proceed character)
DC2	18	X'12'	—	Turns the *FR device ON
DC3	19	X'13'	—	Pause transmission (XOFF or Pause character)
DC4	20	X'14'	—	Turns the *FR device OFF

NOTE: You an use the XON and XOFF parameters when entering COMM to change the handshaking characters that COMM recognizes. These are the default values that are part of the ASCII standard.

The XON and XOFF characters control the transmission of data. If handshaking is ON, when the transmitting device receives an XON character, the device starts transmitting. It continues transmission until it receives an XOFF or Pause character.

If handshaking is ON, when COMM receives an XOFF character, transmission stops. COMM continues to receive characters, but does not transmit until it receives an XON character.

To resume transmission after COMM receives an XOFF character, execute an *CL ON command. To enter an *CL ON command, type (CLEAR)(4) (CLEAR) (:).

You can specify a pause character to force COMM to stop sending data when you transmit that charcter. To specify a pause character, type (CLEAR) (SHIFT) (*) and the character you want to pause on. Do not include the (CLEAR)(:) after the pause character.

For example, you may want to specify (ENTER) as the pause character so that line-at-a-time transmission occurs. COMM pauses at the end of each line and waits until the receiving computer sends an XON character.

The *FR device ON and *FR device OFF control the recording device. They tell the recording device (*FR) to start and stop recording received data. You must create an *FR file with (CLEAR)(6) and (CLEAR)(9) before you use these controls.

(CLEAR)(SHIFT)(=)

Exits to TRSDOS Ready. It does not require any ON or OFF code.

Before COMM stops running, it checks the "Data Received" device (*FR) to see if any open files exist. If there is an open file, COMM closes it before it exits to TRSDOS. This feature prevents you from having unclosed files in your system.

Quick Reference Label

If you are a beginning COMM user, you may find it helpful to make a label containing each key's function and place the labels directly above the keyboard. Label the keys as follows:

Key	Unshifted	SHIFTed
1	*KI	Duplex
2	*DO	Echo
3	*PR	Echo-Linefeed
4	*CL	Accept-Linefeed
5	*FS	Rewind File

6	*FR	Position @ EOF
7	DTD	DCC
8	Menu	Clear Screen
9	ID	8-bit mode
0	Reset	Command
:	On	Handshaking
-	Off	Exit

Logging-On To CompuServe (Available only in U.S.A.)

You can use your computer and COMM to log-on to CompuServe. To log on to CompuServe, you must first buy a Universal Sign-Up Kit (Radio Shack Cat. No. 26-2224). Next, follow these steps:

1. First, use the SET command to SET *CL to COM/DVR (see Appendix I). Then issue the command:

   ```
   SETCOM (WORD=8,PARITY=OFF,STOP=1) (ENTER)
   ```

2. Type:

   ```
   COMM *CL (ENTER)
   ```

3. Now you need to dial CompuServe's number that comes in the Universal Sign-Up Kit Depending on which modem you are using, you either dial the number on a phone or you must enter the commands that cause the modem to dial the number for you. See your modem manual for the correct procedure.
4. After the number is dialed, wait for a "carrier tone" that CompuServe sends to tell you that you are connected.
5. Now press (CTRL)(C) to send a hexadecimal value of 03 to CompuServe.
6. CompuServe prompts you on your video display with:

   ```
   User ID:
   Password:
   ```

   Answer each prompt with the numbers supplied in the Universal Sign-Up Kit and (ENTER).
7. You are now logged-on to CompuServe.

COMMunicating with Bulletin Board Systems

A Bulletin Board System (BBS) is typically a small computer used by individuals, schools, or companies that provides a communication link between its users.

With some TRS-80 Bulletin Board Systems, you can receive graphics characters. For you to be able to accept these graphics, the COM/DVR driver has to be initialized at 8 bits per word (see the SETCOM library command) and you have to use 8-bit mode in COMM ((CLEAR)(SHIFT)(]) followed by (CLEAR)(9)).

COMMunicating with Other Computers

This section shows you how to use COMM to communicate with other computers. The first example describes how a TRS-80 communicates with a mainframe computer. The second example describes how two TRS-80's can communicate.

COMMunicating with a Mainframe

When a TRS-80 communicates with a mainframe computer, in most cases it is not necessary to change the default device or function settings when you enter COMM. Most mainframes operate as the host computer while you operate as a terminal, and the mainframe provides echo functions for you. You must be sure to specify the RS-232-C parameters when setting up the COM/DVR driver to match those expected by the mainframe.

To transfer a file from a mainframe to your TRS-80 computer, use the following procedure:

1. Type in the command which causes the mainframe to list the file, but do not press (ENTER).
2. Specify your receive file by pressing (CLEAR)(6) followed by (CLEAR)(9). Type in the filename in response to the prompt.
3. Press (CLEAR)(6) followed by (CLEAR)(:) to open the receive area of memory. If the file you wish to receive is larger than your available area of memory, you should then press (CLEAR)(7) followed by (CLEAR)(:). This causes the file to be written to the disk as it is being received.
4. Press (ENTER) to start the file listing.
5. When the listing is complete, press (CLEAR)(6) followed by (CLEAR)(-) to turn OFF the *FR and if you have not already done so, press (CLEAR)(7) followed by (CLEAR)(:) to write the file to disk.
6. When the disk write is complete, type (CLEAR)(6) followed by (CLEAR)(0) to turn off DTD and to close the receive file.

To transfer a file from your TRS-80 computer to a mainframe, use the following procedure:

1. Designate the file that you want to send by pressing (CLEAR)(5) followed by (CLEAR)(9) and entering the name of the file in response to the prompt.
2. Turn on the handshake mode by pressing (CLEAR)(SHIFT)(*) followed by (ENTER) (assuming that the line terminating character in your file is (ENTER)).

If the mainframe does not support handshaking, first try to transfer the file without the handshake mode. If this doesn't work, contact the mainframe's computer site and find out how to send files to that mainframe.

3. Open the file at the host end and ready it for receiving information by whatever command process your host requires.

4. Turn on your file send by pressing (CLEAR)(5) followed by (CLEAR)(:).

   Note that one line of your file is transmitted and then your machine pauses. Once the host sends you the XON, the next line of the file is automatically transmitted.

   If you are operating in half-duplex, you may see the entire file displayed without any pauses. The file is being read from your disk and put in an area of memory where it waits to be transmitted.

5. When the transmission is complete, turn off the handshake mode by pressing (CLEAR)(SHIFT)(*) followed by (CLEAR)(-).

6. Close the file at the host end by whatever command process the host accepts. You may then close your file send by pressing (CLEAR)(5) followed by (CLEAR)(0) (which turns off the *FS and closes the file).

   If you want to force the transmission to resume after a line is ended, you may turn the *CL back on by pressing (CLEAR)(4) followed by (CLEAR)(:).

COMMunicating Between Two TRS-80's

When you use COMM to communicate between two TRS-80's, one end has to run on half-duplex ((CLEAR)(SHIFT)(!) followed by (CLEAR)(:)) and echo ((CLEAR)(SHIFT)(") followed by (CLEAR)(:)). If files are to be sent and received, the RECEIVING end should run half-duplex and echo.

To transfer files between two TRS-80's, use one of the following two methods. Use Method A if you are operating above 300 baud. Use Method B if you are operating at 300 baud.

Method A

1. The sending end presses (CLEAR)(5) followed by (CLEAR)(9) and enters the name of the file to be sent.

2. The receiving end presses (CLEAR)(6) followed by (CLEAR)(9) and enters in the name of the file to be received. Turn the dump-to-disk (DTD) OFF by pressing (CLEAR) (7) followed by (CLEAR)(-). This stores the file in memory as it is received.

If the sending end supports XON/XOFF handshaking, then you should turn HANDSHAKE ON by pressing (CLEAR)(SHIFT)(*) followed by (CLEAR)(:).

3. When both ends are ready, the receiving end presses (CLEAR)(6) followed by (CLEAR)(:), after which the sending end presses (CLEAR)(5) followed by (CLEAR)(:).

   If your free area of memory decreases to less than 2K during receipt of the file, a warning message is issued and an XOFF is automatically sent to the sending end.

   Transmission from the sender should cease. Once it does, dump the receive area of memory to disk by turning on DTD by pressing (CLEAR)(7) followed by (CLEAR)(:).

   You can observe the increase in available memory space by displaying a menu as the area of memory is written to disk. Once ample space is available, turn off the DTD by pressing (CLEAR)(7) followed by (CLEAR)(–).

   You can then manually restart the sender's file by transmitting an XON from your keyboard with (CTRL)(Q).

4. The receiving end presses (CLEAR)(6) followed by (CLEAR)(–) when it has received all of the file. The last receive area of memory should be dumped to disk by turning on DTD ((CLEAR)(7) followed by (CLEAR)(:)).

   The sending end presses (CLEAR)(5) followed by (CLEAR)(-) and then (CLEAR)(5) followed by (CLEAR)(0).

5. When the receiving end has finished writing the information to the disk, close the file by resetting the *FR ((CLEAR)(6) followed by a (CLEAR)(0)). This performs an *FR OFF and a DTD OFF, and it closes the file just received.

Method B

1. The sending end presses (CLEAR)(5) followed by (CLEAR)(9) and enters in the name of the file to be sent.

2. The receiving end presses (CLEAR)(6) followed by (CLEAR)(9) and enters in the name of the file to be received.

   The dump-to-disk (DTD) must be turned ON by pressing (CLEAR)(7) followed by (CLEAR)(:) . Check to see if it is already ON by displaying a menu ((CLEAR)(8)) and noting if an asterisk is displayed beneath its key.

3. When both ends are ready, the receiving end presses (CLEAR)(6) followed by (CLEAR)(:). The sending end then presses (CLEAR)(5) followed by (CLEAR)(:) to turn ON the receive and send files.

4. When the receiving end has received all of the file and it is written to the disk, close the file by resetting the *FR. Press (CLEAR)(6) followed by (CLEAR)(0). This performs an FR OFF and a DTD OFF, and it closes the file just received. The sending end then presses (CLEAR)(5) followed by (CLEAR)(0).

Technical Information

This section describes some of the more technical aspects of COMM operation. This information allows you to predict how COMM will perform during higher speed I/O operations.

Main memory usage

COMM uses all available memory below the top of memory mark (HIGH$) for dynamic buffering of device I/O. You can see or set this value with the MEMORY command.

The amount of buffer space devoted to each logical device dynamically expands and shrinks according to how quickly data is sent to a device and how fast the device can process the data it receives. Each buffer is essentially a variable length First-In, First-Out (FIFO) storage compartment.

The amount of free space available for the buffers is noted in the bottom line of the menu display. When this free space shrinks to less than 2K (2048 characters), a warning message is displayed and an XOFF is automatically sent to the communications line (*CL).

This function is useful when you are receiving a file from a system that supports handshaking. (The (CLEAR)(SHIFT)(*) command describes the supported handshaking.)

Break commands

COMM generates a modem break (long space) when you press the (BREAK) key. A modem break is used on many mainframe systems to indicate you want to abort a function that is occurring at the other computer.

However, for small computers, detecting a modem break is more difficult, so you have to select a control character to be treated as a "break" command.

To transmit a break character to another computer, press (CTRL)(C) if the other computer is a Model II, 12 or 16. Press (CTRL)(A) if the other computer is a Model I or III.

Escape code sequences

Some systems transmit control codes to indicate that a cursor movement or action is to be performed. Many systems have adapted a two-character sequence, which does not perform the intended function in COMM.

If you are working with one of these systems, you should contact the operators of the other system and ask if there is a way to prevent these control sequences from being sent to your system.

Some systems support several different types of terminals and computers, so with a little experimenting, you should be able to find a terminal setting that suits your needs.

Receiving large files from another system

If you receive files that won't fit into memory in one piece, you may have to use handshaking to reduce the possibility of losing data.

CONV (CONV/CMD)

Utility

CONV [*partspec* **:** *-partspec*]:*source drive* [:*destination drive*][(*parameters*)]

Allows you to move (convert) data files from a TRSDOS 1.3 (Model III) diskette onto a TRSDOS Version 6 formatted diskette.

This command requries two floppy disk drives.

Use this command with data or BASIC ASCII files. TRSDOS 1.3 application programs will not work on TRSDOS Version 6. To use TRSDOS 1.3 programs on your computer, start up your system with a TRSDOS 1.3 system diskette in Drive 0.

If you specify *partspec*, CONV moves the files that match *partspec*. If you specify *partspec* preceded by a hyphen, CONV moves the files that do not match *partspec*. If you include the drive number with *partspec*, you must include the colon (:).

The *parameters* are:

VIS moves visible files
INV moves invisible files
SYS moves system files
NEW moves only those files that do not already exist on the destination disk.
OLD moves only those files that already exist on the destination disk.
QUERY=NO specifies that you are not to be questioned before each file is moved to the destination disk.
DIR displays a short directory of a TRSDOS 1.3 disk. If you do not specify *destination drive*, a short directory is displayed. If you specify DIR, no files are moved.

If you don't specify VIS, INV, or SYS, TRSDOS moves all three types of files.

The TRSDOS 1.3 disk must be a non-limited backup disk. Some programs such as SCRIPSIT and VISICALC are limited backup disks.

If you have data files on a TRSDOS 1.3 limited backup disk, you must COPY these files (under TRSDOS 1.3) to a non-limited backup disk before you can CONVert them.

The *source drive* cannot be Drive 0.

When you specify a *partspec*, only those files matching the partspec are moved to the destination disk.

When you do not specify the QUERY=NO parameter, you are questioned before each file is moved. Answer the prompt by pressing:

(Y)	to copy to file.
(N) or (ENTER)	to bypass the file and show the next one.

Caution: Do Not move BASIC/CMD or any other existing TRSDOS system files.

Error Conditions

If you attempt to copy certain limited Model III TRSDOS 1.3 diskettes with the CONV utility, TRSDOS displays a "Cannot CONV Protected Diskette" error message.

Examples

```
CONV :2 :1 (ENTER)
```

moves all files from Drive 2 onto Drive 1. You are questioned before each file is moved. If the file already exists in Drive 1, you are asked again before it is copied.

```
CONV :1 :0 (VIS,Q=NO) (ENTER)
```

moves all visible files from Drive 1 onto Drive 0. You are not questioned before each file is moved.

```
CONV :2 :0 (NEW) (ENTER)
```

moves only those files from Drive 2 that do not already exist on Drive 0.

```
CONV $$$DATA:1 :2 (OLD) (ENTER)
```

moves any file whose filename is seven or eight characters long, the 4th through 7th characters are DATA, and that already exists on Drive 2. You are questioned before each file is moved.

```
CONV :1 (DIR) (ENTER)
```

displays the directory of the TRSDOS 1.3 disk in Drive 1.

```
CONV :1 (INV,DIR) (ENTER)
```

displays the invisible files of the TRSDOS 1.3 disk in Drive 1.

COPY

COPY *source* [TO] *destination* [(*parameters*)]	Utility

Copies the *source* to the *destination*.

Source and *destination* can be a filespec or a devspec. *Destination* can also be a drive number.

The parameters are:

LRL=*number* specifies the logical record length (1 to 256) for *destination*. If you omit the *number*, *destination* will have the same LRL as *source*.

CLONE=NO specifies that *destination* is not to have the attributes of *source*.

ECHO causes any character copied from a devspec to be printed on the screen.

X allows a single drive copy.

The LRL parameter lets you restructure files to make them compatible with other programs. It is also useful when converting a source file from one format to another.

If you wish to append two files with different LRLs, this parameter can be used to make the LRLs match. If LRL is not specified, it defaults to the LRL of *source*.

If CLONE=NO is specified, the directory entry as well as the contents of *source* copies to *destination*. The owner and user passwords are copied, along with the assigned protection level, the visibility in the directory, the create flag, the last written-to date, and the modified status of the file.

If CLONE=NO is specified, the system date becomes the last written-to date for the destination file. If an existing destination file was copied over, the attributes of the destination file (except for the date) are unchanged. If the COPY command creates a new file, any password included becomes both the user and owner password of the destination file and the file's Mod Flag is set. The destination file is visible, even if the source file was invisible. See the ATTRIB library command for more information on file attributes.

If you omit an extension for the destination filespec, TRSDOS creates a destination file that has the same extension as the source. To override this, add a slash (/) to the end of the *destination* when entering the command.

Examples

```
COPY TEST/DAT TO :1 (ENTER)
```

searches the disk drives to find TEST/DAT and copies it to Drive 1.

```
COPY TEST/DAT.PASSWORD:0 TO :1 (ENTER)
```

copies the protected file TEST/DAT.PASSWORD from Drive 0 to Drive 1. All parts of the destination file, including the password, are the same as those of the source file.

```
COPY TEST/DAT:0 TO MYFILE:1 (ENTER)
```

copies TEST/DAT on Drive 0 to MYFILE/DAT on Drive 1. Since the destination filespec does not contain an extension, it defaults to /DAT to match the source.

```
COPY DATA/NEW:0 TO /OLD:0 (ENTER)
```

copies DATA/NEW on Drive 0 to DATA/OLD on Drive 0. Since the destination filespec does not contain a filename, it defaults to DATA to match the source.

```
COPY TEST/DAT:0 TO TEST/DAT.CLOSED:1
(CLONE=NO) (ENTER)
```

copies TEST/DAT from Drive 0 to Drive 1 and assigns the user and owner passwords CLOSED. To assign a password to a destination filespec, the CLONE parameter must be turned off.

```
COPY DATA/V56:0 TO DATA/V28:1 (LRL=128) (ENTER)
```

copies DATA/V56 on Drive 0 to DATA/V28 on Drive 1. The LRL of DATA/V28 is set to 128.

```
COPY *KI TO *PR (ECHO) (ENTER)
```

copies from the keyboard to the printer. As keys are pressed, they are sent to the line printer. The keystrokes are visible on the video because the ECHO parameter is specified. Pressing (CTRL)(SHIFT)(@) or (BREAK) terminates the copy.

When copying from devspec to devspec, it is very important that all devices specified be assigned and active in the system. Any routing or setting affecting the devices may affect the copy.

It is *very* important to be aware that you can generate non-ending loops that lock up the system when copying between devices. Be sure to have a good understanding of this type of copy before you use it.

```
COPY *KI TO KEYIN/NOW:0 (ENTER)
```

sends the keystrokes entered from the keyboard to the file KEYIN/NOW on Drive 0. If the file already exists, it is written over. To view the characters as you type them, use the ECHO parameter. Pressing (CTRL)(SHIFT)(@) terminates the copy.

```
COPY TEST/DAT.SECRET:0 (X) (ENTER)
```

copies TEST/DAT.SECRET from one diskette to another.

The destination file TEST/DAT.SECRET is visible, and its owner and user passwords are set to SECRET.

When you use the (X) parameter, a TRSDOS system diskette is not required in the copy if the proper system modules (1, 2, 3, and 4) are loaded into memory (see the SYSTEM (SYSRES) library command).

During the copy, the following disk swap prompts are repeated until the copy is complete. The prompts are:

```
Insert SOURCE disk <ENTER>
```

This refers to the diskette containing the file to be copied.

```
Insert SYSTEM disk <ENTER>
```

The refers to a TRSDOS system diskette containing the same version of TRSDOS that you started the system with. If system modules 1, 2, 3, and 4 are loaded into memory, press (ENTER) at this prompt.

```
Insert DESTINATION disk <ENTER>
```

This refers to the diskette to receive the file. This prompt may appear twice in a row. The diskette must have a different Disk ID (disk name, master password, or date) from the source diskette. (If it is a system diskette, use ATTRIB if you need to change its Disk ID.)

You cannot use the (X) parameter in copies involving logical devices.

Sample Use

Every time you update a file, use COPY to make a duplicate file on another disk. This protects you from having to re-enter the entire file if the disk is ever damaged.

Use COPY to reduce file fragmentation. File fragmentation exists when there is not enough contiguous space on a disk to store the file. TRSDOS uses small areas of the disk where they are available. Fragmentation increases the amount of time required to access the data in the file. To reduce fragmentation, COPY the file to a disk that has enough contiguous space for the file. The FREE command displays the amount of free space on a disk. (See FREE and DIR for more information on file fragmentation.)

To RENAME a file on the same disk, use RENAME, not COPY.

Error Conditions

If you include the X parameter with a COPY command in a JCL file, TRSDOS displays an "Invalid command during DO processing" error message. You cannot change disks during JCL processing.

If the source and destination disks are the same and you execute a COPY command with the X parameter, TRSDOS displays a "Source and destination disks are the same" error message. You can omit the X parameter if the source and destination disks are the same.

CREATE

Advanced Programmer's Command

CREATE *filespec* [(*parameters*)]

Creates a file named *filespec* and pre-allocates space for its future contents.

You can use CREATE to prepare a file which will contain a known amount of data. This usually speeds up file write operations. File reading is also faster, since pre-allocated files are usually less segmented or dispersed on the disk — requiring less motion of the read/write mechanism to locate the records.

The smallest unit of space TRSDOS allocates for a file is one granule. A granule is one or more 256 byte sectors. The size of a granule varies depending on the type of disk you are using. On double-density diskettes there are six 256 byte sectors or 1.5 K in a granule. Use the FREE command to see how large a granule is on a disk. See the FREE command for more information.

If the size of a file requires more than one granule, but less than two granules TRSDOS allocates 2 granules. TRSDOS cannot allocate a portion of a granule. To determine how many granules will be allocated for a file of a specific length, use one of these formulas for double density diskettes.

(LRL * REC / 256) / 6   or   SIZE / 1.5

When a file is CREATEd, TRSDOS does not recover unused space at the end of the file (each time you finish using it). If you exceed the created size, TRSDOS allocates extra space for your file as you write to it.

The parameters are:

REC=*number* assigns the specified *number* of fixed-length records to the file.

LRL=*number* assigns *number* as the record length of *filespec*. *number* can be from 1 to 256. If you omit this parameter, the record length defaults to 256.

SIZE=*number* allocates disk space to the file as *number* (in K).

Note: You may not use SIZE if you include LRL or REC. You may only use SIZE with files that contain 256-byte records or to CREATE new files that contain 256-byte records. To increase the size of a file that does not contain 256-byte records, use LRL and REC to specify the new size as a larger number of records.

(For more information about record lengths and types, see "Disk Files" in the *Model 4/4P Technical Reference Manual*.)

CREATE also lets you permanently assign additional space to a file that already exists. Use the appropriate parameters for the new file size.

Examples

```
CREATE NEWFILE/DAT:0 (LRL=128,REC=100) (ENTER)
```

creates a file named NEWFILE/DAT on Drive 0 and allocates space for one hundred 128-byte records.

```
CREATE GOOD/DAT (REC=50) (ENTER)
```

creates a file named GOOD/DAT on the first available drive and allocates space for fifty 256-byte records.

```
CREATE INVENT/DAT (SIZE=20) (ENTER)
```

increases the size of the already existing file, INVENT/DAT, to 20 K-bytes. INVENT/DAT must contain 256-byte records to use the SIZE parameter. If the records are not 256-byte, use the LRL and REC parameters to increase the size.

Error Conditions

If the SIZE or REC parameters specify less disk space than is already allocated for an existing file, TRSDOS displays a "File exists larger" error message. The size and contents of the file are not changed.

If you omit *drive*, TRSDOS attempts to create the file on the first available drive. If there is not enough space on that disk, TRSDOS displays a "Disk space full" error message. Use the FREE command to display available disk space. Try the CREATE command again, specifying a drive that contains enough free space for the file that you are creating.

Sample Use

Suppose you are going to store personnel information on no more than 250 employees, and each data record will look like this:

Name (Up to 25 letters)
Social Security Number (11 characters)
Job Description (Up to 92 characters)

Your records would need to be 25 + 11 + 92 = 128 bytes long.

You could create an appropriate file with this command:

```
CREATE PERSONNL/TXT (REC=250,LRL=128) (ENTER)
```

Once created, this pre-allocated file would allow faster writing than would a dynamically allocated file, since TRSDOS would not have to stop writing periodically to allocate more space (unless you exceed the pre-allocated amount by adding more than 250 employees).

DATE

DATE [*mm/dd/yy*]	**Command**

Sets or displays the current system date.

When you start up your computer, you set the current system date. TRSDOS uses that date when creating and accessing files, making backups, and formatting. You can change the system date with the DATE command. If you omit *mm/dd/yy*, TRSDOS displays the current system date.

mm (month) is a 2-digit number in the range 01 to 12.

dd (day of the month) is a 2-digit number in the range 01 to 31. *dd* must be a valid day of the month specified. For example, you cannot specify *dd* as 31 when you specify 04 as the month. April does not have 31 days.

You must include leading zeroes for month and day.

yy (year) is a 2-digit number in the range 80 to 87.

You can use any of the characters in the ASCII range 32 (X'20') through 39 (X'27') , 41 (X'29') through 47 (X'2F') and ASCII 58 (X'3A'), to separate month, day, and year. See Appendix C for a complete list of the ASCII character codes.

Error Conditions

If you specify a value outside the valid ranges for *mm, dd,* and *yy,* or if you specify an invalid separator between those values, TRSDOS displays a "Bad Date format" error message.

When you execute a SYSTEM (DATE=NO) command, TRSDOS does not store the current system date. Any attempt to display the date results in a "Date not in system" error.

Examples

```
DATE (ENTER)
```

displays the current date, such as:

```
Fri, Oct  8, 1982
```

for Friday, October 8, 1982.

```
DATE 10/09/82 (ENTER)
```

resets the date to October 9, 1982 and displays the new date.

Advanced Programmer's Command

DEBUG [([*switch*] [,] [*EXT*])]

The DEBUG command sets up the debug monitor, which allows you to enter, test, and debug machine-language programs.

The *switches* are:

ON	turns on DEBUG
OFF	turns off DEBUG

If *switch* is not specified, ON is assumed.

EXT	specifies the extended debugger

EXT, ON, and OFF can be abbreviated to E, Y, and N.

Once you have turned on DEBUG, you automatically enter the debug monitor whenever you do one of the following:

1. Press the (BREAK) key (provided (BREAK) is enabled)
2. Load and execute a user program (as long as the file's protection is not execute only)

You can also automatically activate the debugger by holding down the (D) key while the system is booting.

While in the DEBUG monitor, you can enter any of a special set of single-key commands to study how your program is working (as detailed under Command Description below).

EXT loads a separate block of the system debugger into high memory. This block contains additional functions and commands. While DEBUG is on, TRSDOS automatically protects this area of memory from being overlaid by BASIC or other user programs.

If you execute a program with execute-only protection, and you fail to supply the OWNER password, DEBUG is disabled while that program is running.

Examples

```
DEBUG (ENTER)
```

turns on the standard DEBUG and waits for it to be activated.

```
DEBUG (EXT) (ENTER)
```

turns on extended DEBUG (loads it into high memory) and waits for it to be activated.

```
DEBUG (OFF) (ENTER)
```

turns off standard or extended DEBUG.

```
DEBUG (OFF,EXT) (ENTER)
```

Turns off DEBUG and attempts to reclaim the high memory occupied by the extended debugger. If another program is loaded in high memory below (after) the extended debugger is loaded, the space used by the debugger cannot be reclaimed without resetting the system.

To enter the monitor when DEBUG is on, type:

```
filespec (ENTER)
```

TRSDOS loads *filespec*. If you specify the OWNER password, and if the protection level is READ or higher, TRSDOS transfers control to DEBUG.

Following is a sample display of the debugger screen.

The debug display contains information about the Z-80 microprocessor registers. The display is set up in the following manner:

> The register pairs are shown along the left side of the display, from top to bottom. The current contents of each register pair are shown immediately to the right of the register labels.
>
> The AF and AF' pairs are followed by the current status of the flag registers to the right of the register contents. The other register pairs are followed by the contents of the 16 bytes of memory they are pointing to. The contents are shown in both hexadecimal and ASCII representations. Non-displayable ASCII characters are represented by periods.

The PC register shows the memory address of the next instruction to be executed. The display to the right of that address shows the contents of that address and the next 15 addresses.

The bottom four lines of the screen show the contents of the memory locations indicated by the address at the left of each line. These locations vary depending upon which command is used.

Command Descriptions

When the DEBUG screen is displayed, you can enter one of the following single-key commands.

You must enter all numerics, addresses and quantities, as hexadecimal values. If you make a mistake entering these hexadecimal values, simply type the correct value before you press (SPACEBAR). DEBUG ignores all but the last four digits in an address and all but the last two digits of a byte. For example, if you want to enter some data at address 6789 and you type 6780, type the correct address before you press (SPACEBAR).

```
H67806789 (SPACEBAR)
```

A (ASCII Modify)

A*address*

Enter the above command to modify *address*. If the contents of address are already on the display, vertical bars appear around the byte being modified.

After you enter *address*, press (SPACEBAR). The address and its contents appear in the lower left corner of the screen. To modify the byte, type the new character. DEBUG moves to the next byte and allows you to modify it.

In addition to typing a new character, you may also press:

- (SPACEBAR) to retain the value of the current address and move to the next address.
- (ENTER) to exit from the A command.

Note: You cannot use the X command to cancel an incorrect A command after you have entered an address to modify. If you press (X) after you have entered an address, DEBUG stores the character X at the current address. You must use (ENTER) to exit the A command.

To store an ASCII space character, (X'20'), or a carriage return (X'0D'), you must use the H command and enter a '20' or a '0D' at the address of the space.

If you do not specify *address*, TRSDOS uses the current "memory modification address" (shown by the vertical bars).

Example:

`AD004` (SPACEBAR)

displays the D004 and the character stored at D004 in the lower left corner of the screen. Type the new character. Press (ENTER) to exit from the A command or more characters to change the next consecutive addresses.

B (Move Block of Memory)

B*starting address*,*destination address*,*number of bytes*

Enter the above command line to move a block of memory from *starting address* to *destination address*.

Always specify a non-zero *number of bytes*. If you enter *number of bytes* as 0, TRSDOS moves a block of 65,535 bytes to *destination address*, which could cause the system to lock up and require resetting.

Example:

`B3E04,4E34,14E` (ENTER)

moves the 14E-byte blcck of memory from 3E04 to 4E34.

C (Call Instruction)

Press (C) to single-step through the instructions pointed to by the PC register. If a call instruction is encountered, the routine that it calls is executed and is not single-stepped. DEBUG pauses after the subroutine returns to the instruction following the call.

D (Display)

D*address*

Enter the above command line to display memory beginning at *address*.

Example:

`DE404` (ENTER)

displays memory beginning at address E404.

F (Fill Memory)

F*first address*,*last address*,*byte*

Enter the above command line to fill the block of memory from *first address* to *last address* with the value *byte*.

Example:

`F3D08,3E14,00` (ENTER)

fills the block of memory from 3D08 to 3E14 with the value 00.

G (Go to an Address and Execute)

G*address*,*breakpoint1*,*breakpoint2*

Enter the above command line to begin execution at *address*. If *address* is omitted, execution begins at the PC address.

breakpoint1 and *breakpoint2* are optional breakpoint addresses where execution stops. The system removes them when you return to debug.

Example:

```
GE0FF,F001,F201 (ENTER)
```

begins execution at E0FF. Stops execution at breakpoint addresses F001 or F201.

H (Hex Modify)

H*address*

Enter the above command to modify *address*. If the contents of address are already on the display, vertical bars appear around the byte being modified.

After you enter *address*, press (SPACEBAR). The address and its contents appear in the lower left corner of the screen. To modify the byte, type the hexadecimal value. DEBUG moves to the next byte and allows you to modify it. You may also press:

- (SPACEBAR) to modify the byte and move to the next address.
- (ENTER) to modify the byte and exit from the H command.
- (X) to exit from the H command without modifying the current byte.

Note: TRSDOS stores the new data as soon as you press (SPACEBAR) or (ENTER).

If you do not specify *address*, TRSDOS uses the current "memory modification address" (shown by the vertical bars).

Example:

```
HD004 (SPACEBAR)
```

causes address D004 and the current byte value to appear in the lower left corner of the screen. TRSDOS allows you to enter the new byte value. Then press (SPACEBAR), (ENTER), or (X) to continue.

I (Single-Step Execution)

Press (I) to single-step through the instructions pointed to by the PC register. This command is identical to the C command except that any calls encountered are stepped through instruction by instruction. (Note

that RST 28H, RST 30H, and RST 38H instructions automatically convert the I command to a C command.)

J (Jump)

Press (J) to increment the program counter (PC) by 1.

O (Return to TRSDOS Ready)

Press (O) to return to TRSDOS. DEBUG is not turned off. (Use the DEBUG (OFF) command to turn DEBUG off.)

Q (Port)

There are two kinds of ports — input and output. You read an input port and you write to an output port.

Q*port*

Enter the above command line to read the byte at *port* and display its value. There are 256 input ports (00 - FF).

Example:

```
Q45 (ENTER)
```

displays the value of port 45 in the lower left corner of the screen.

Q*port,byte*

Enter the above command line to write the value of *byte* to *port*. There are 256 output ports.

Example:

```
Q45,04
```

writes the byte value of 04 to port 45.

R (Register Pair)

R*register pair code contents*

Enter the above command line to change the specified register pair's contents to the new *contents*. There must be a space between *register pair* code and *contents*.

The *register pair* codes are:

AF	for	AF	AF'	for	AF'
BC	for	BC	BC'	for	BC'
HL	for	HL	HL'	for	HL'
DE	for	DE	DE'	for	DE'
IX	for	IX			
IY	for	IY			
SP	for	SP			

Example:

```
RBC 3D01
```

changes the contents of register pair BC to the value 3D01.

S (Full Screen Mode)

Press (S) to change the monitor format from the register display mode to full screen mode. The full screen mode displays a page of memory (256 bytes) that contains the current display address. (See the D command.)

U (Update)

Press (U) to constantly update the display and to show any active background tasks. To cancel this command, hold down any key for several seconds.

X (Return)

Press (X) to return the display to the normal register display mode.

; (Advance Memory)

Press (;) to advance the memory display 64 bytes in the register mode and 256 bytes in the full screen mode.

– (Decrement Memory)

Press (-) to decrement the memory display by 64 bytes in the register mode and 256 bytes in the full screen mode.

Disk Read/Write Utility

Lets you read or write to a specified block of memory. The command line is:

> *disk drive,cylinder,starting sector,operation,address,number of sectors*

address is the starting address in memory where the information read from the disk is to be placed, or where information written to the disk is to be taken from.

Specify operation as: R for Read, W for Write, or * for a Directory Write.

If you do not specify *cylinder*, the system uses the directory track. If you do not specify *starting sector*, the system starts with sector 0. If you do not specify *number of sectors*, the system reads the whole cylinder.

If an error occurs during a disk function, the error number appears on the screen surrounded by asterisks. Hold down the (ENTER) key to abort the disk function.

Example:

`2,0,0,R,6000,2` (ENTER)

reads into memory (beginning at address X'6000') sectors 0 and 1 of cylinder 0 from the disk in Drive 2. This block of memory is displayed on the monitor in the full screen mode.

Extended Command Descriptions

The following commands are available only with the extended debugger.

E (Enter Data)

E*address*

Enter the above command to enter data directly into memory beginning at *address*. The contents of *address* are displayed and you can then type in two hex characters to replace the current contents. After typing the byte, press:

- (SPACEBAR) to modify the byte and move to the next address.
- (ENTER) to modify the byte and exit from the E command.
- (X) to exit from the E command without modifying the byte.

If you do not specify *address*, TRSDOS uses the current memory modification address (shown by the vertical bars).

Note: TRSDOS stores the new data as soon as you press (SPACEBAR) or (ENTER).

L (Locate)

L*address,byte*

Enter the above command to locate the first occurrence of *byte*, starting the search at *address*.

If you don't specify *address*, DEBUG uses the current memory modification address (shown by the vertical bars). If you don't specify *byte*, DEBUG uses the last byte given in a previous L command.

Example:

`L470E,0D` (ENTER)

searches for the first occurrence of 0D after address 470E.

N (Next Load Block)

Enter the above command to position the vertical bars to the next load block. This command is used to move logically through a block of memory that has been loaded directly from disk using DEBUG.

The load block type byte is a byte at the beginning of every block. Before you can use this command you must position the vertical bars over the load block type byte. To do so, use the H command by typing:

H*address* (ENTER)

Example:

Position the vertical location bars over the beginning byte of a load block and type:

N (ENTER)

DEBUG advances to the beginning byte of the next load block.

P (Print)

P*first address,last address*

Enter the above command line to print out the block of memory from *first address* to *last address*.

Example:

PFC80,FC90 (ENTER)

prints out the block of memory from FC80 to FC90 in the following format:

aaaa bb bb . . bb cccccccccccccccc

aaaa	represents the current address
bb bb . . . bb	represents 16 locations in hex notation
cccc	represents the ASCII equivalents of the 16 hex locations

T (Type ASCII)

T*address*

Enter the above command line to type ASCII characters directly into memory, starting at *address*. If you omit *address*, DEBUG uses the current memory modification address (shown by the vertical bars).

Example:

TCB01 (SPACEBAR)

displays the address CB01 and its current contents in ASCII code. If the contents of the address are out of the ASCII character range, then a period is displayed.

DEBUG then prompts you to enter the new ASCII contents for CB01. Type:

```
A
```

to enter the hex value for A, which is 41, in address CB01.

Pressing (SPACEBAR) advances memory one byte without changing its contents. DEBUG continues to prompt you to add ASCII values until you press (ENTER) to exit the command.

V (Compare)

V*first address,second address,length*

Enter the above command line to compare a block of memory beginning at *first address* to the block of memory beginning at *second address*. The compare is for the specified *length* in bytes (X'0001' – X'FFFF').

Example

```
VCB00,EF02,45 (ENTER)
```

compares a 45-byte long block of memory beginning at CB00 to a 45-byte long block of memory beginning at EF02. The first byte of the block of memory beginning at CB00 that does not match is displayed as the first byte of memory in the DEBUG monitor. The corresponding byte in the block of memory beginning at EF02 becomes the current memory modification address that is used by the H, A, E, and T commands.

W (Word)

W*address,word*

Enter the above command to search memory for *word*, beginning at *address*. *word* must be in the least significant byte, most significant byte format.

If you do not specify *address*, DEBUG uses the current memory modification address. If you do not specify *word*, DEBUG uses the last word given in a previous W command.

Example:

```
WAB06,3412 (ENTER)
```

searches memory for word (1234) beginning at address AB06. The address where *word* is found is displayed in the DEBUG monitor with the vertical location bars positioned one byte before it.

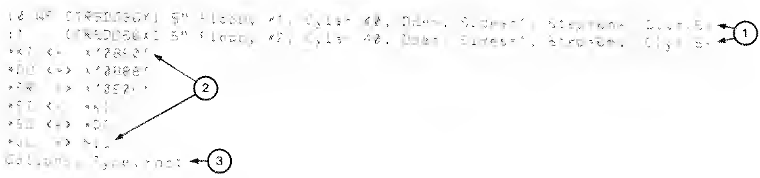
DEVICE [(*parameters*)]	**Advanced Programmer's Command**

Displays the status of the drives, the options selected, and the data paths for the logical devices that have been set, routed, or linked.

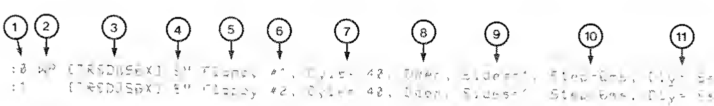
It also logs in disks in the available disk drives.

The parameters are:

D=NO	suppresses the drive portion of the display. Any new drives or disks are not detected.
B=YES	enables the logical device portion of the display.
S=NO	suppresses the options status portion of the display.
P=YES	duplicates the display to the printer.



1. The DRIVE section shows the current configuration of the disk drives.
2. The DEVICE section shows the devices (displayed when B=YES).
3. The STATUS section displays the status of some user selected options.



1. The number of the drive accessed (the logical drive number). (See the SYSTEM command.)
2. The write protect status assigned to the drive by the SYSTEM (DRIVE=,WP=) command. A disk can also be write protected by placing a foil tab over the write-protect notch on the diskette.
   WP=Write Protected
3. Disk name.
4. Disk size.

5. Type of drive — either floppy or hard.
6. For floppy disk systems, the physical location of the drive.
   1 - lower
   2 - upper
   4 - middle of the disk expansion cable
   8 - end of cable
7. The number of cylinders specified when the disk in the drive was formatted.
8. The data density of the disk accessed in the drive.
   Dden = Double density
   Sden = Single density
   Fixed = hard disk
9. The number of sides the floppy diskette has available for storage.
   1 = One side
   2 = Two sides
10. The step rate of the floppy disk drive (in milliseconds). The step rate is the speed at which the disk drive head is moved from cylinder to cylinder.
11. The delay time for accessing a 5″ floppy disk. This is the amount of time the system waits after starting the drive motor before it attempts to access the disk.

NOTE: The Step rate and Delay time are preset for the system. See the SYSTEM command to change these values.

```
*KI  < =   X'08F0'
*DO  < = >X'0B88'
*PR    = >*L0| *TD & = > X'0E0F'
*SI  < = *KI
*SO  < = >*DO
*JL    = >Nil
*FF  < # >[Inactive]X'0FFE'
*TD  < = >PRINT/TXT:0
*L0    = >X'0E0F'
```

In the logical device portion of the device table:

```
<=      indicates an input device
=>      indicates an output device
<=>     indicates a device capable of input and output
#       indicates a filter
|       indicates a link
```

If you add a driver or filter to the default system, the DEVICE command shows the address where a device transfers control to its driver or filter. If more than one filter or driver is associated with the same device, the first driver's address is displayed. It also shows the interaction between devices and/or files.

Options: Type, Fast
System modules resident: 1, 2, 4

The options line displays the active system options. System options are usually established with the FILTER, LINK, ROUTE, SET, SPOOL, and SYSTEM library commands. The options are:

Fast/Slow — indicates the system speed. Fast indicates that the system is running at 4MHz. Slow indicates that the system is running at 2MHz. The slow speed is the Model III system speed. Timing loops in TRSDOS Version 1 programs may require the slower speed for the programs to function properly under TRSDOS 6. The default value for speed is Fast. See the SYSTEM command for additional information.

Forms — indicates that the FORMS filter is resident, and may be active or inactive. The default for FORMS is Off. See the SET, FILTER, and FORMS commands, and Appendix I for additional information.

Graphic — indicates that during screen prints ((CTRL) (:)) the printer has the capability to print graphics characters. Graphics characters are ASCII values greater than X'7F'. You must have a graphics printer to print these characters. The default value for Graphic is off. When Graphic is off during screen prints, the printer prints all characters having a value larger than X'7F' as periods (.).

KSM — indicates that the Keystroke Multiply Filter (KSM) is resident, and may be active or inactive. The default for KSM is off. See the SET and FILTER commands, and Appendix I for additional information.

Memdisk — indicates that the Memdisk utility is active. The default value for Memdisk is Off. See the SYSTEM command and Appendix I for additional information.

Smooth — causes TRSDOS to disable interrupts when reading data from a floppy disk. Smooth increases disk access speed. However, the type-ahead function depends on interrupts occurring at regular intervals. Disabling the interrupts can cause a loss of keystrokes during disk I/O. When SMOOTH is active, the time-of-day clock is not as accurate. It can also result in lost RS232 characters when DTD is on in COMM. The default value for SMOOTH is On. See the SYSTEM command and COMM for additional information.

Spooler	indicates that the device spooler is buffering text being sent to the line printer or the RS-232-C communications line. The default for the spooler is Off. See the SPOOL command or Appendix I for more information on activating the device spooler.
Type	indicates the type-ahead function is buffering keyboard strokes until the system is ready for them. This allows you to type in information before the system prompts you for it. If you disable the type-ahead function, programs run slightly faster. The default for the Type is On. See the SYSTEM command for more information on the type-ahead function.
Verify	indicates that the system is verifying each sector of data as it is written to disk. An error message appears if the system cannot read the data. Although verify slows disk I/O slightly, it can save time in the future if there is something wrong with the data. The default value for Verify is Off. See the VERIFY command for additional information.

The system line displays the resident system overlays. See the SYSTEM (SYSRES=) library command.

Examples

```
DEVICE (ENTER)
```

displays the device table.

```
DEVICE (D=NO) (ENTER)
```

displays the TRSDOS options, and turns the drive portion of the display off. Any new drives or disks are not detected.

```
DEVICE (B=YES) (ENTER)
```

enables the device portion of the device table, and displays the entire table.

```
DEVICE (S=NO) (ENTER)
```

displays the drive table, and turns the options status portion of the display off.

```
DEVICE (P) (ENTER)
```

displays the drive portion on the display, and sends it to the printer.

DIR

DIR [***partspec*** **¦** ***-partspec***][:][***drive1***][-][:][***drive2***]
[**(*parameters*)**]

Command

Displays the directory for one or more drives.

The DIR command displays complete directory information for the files on one drive or a range of diskettes.

If you specify *partspec*, DIR displays only the filenames that match *partspec*. If you specify *partspec*, preceded by a hyphen (-), DIR displays all filenames that do not match *partspec*. If you include a drive number with *partspec*, you must include the colon (:).

Colons are optional in the syntax of the DIR command except when:

- You specify partspec with a drive number.
- You include a colon for *drive1*, you must omit the colon for *drive2*.

If you omit the drive numbers, DIR displays the filenames on all enabled drives. You can include the hyphen to specify a range of drive numbers. If you specify:

drive1-drive2	displays the directory for diskettes in *drive1* through *drive2*.
drive1-	displays the directory for all drive numbers equal to or greater than *drive1*.
-drive2	displays the directory for Drive 0 through *drive2*.

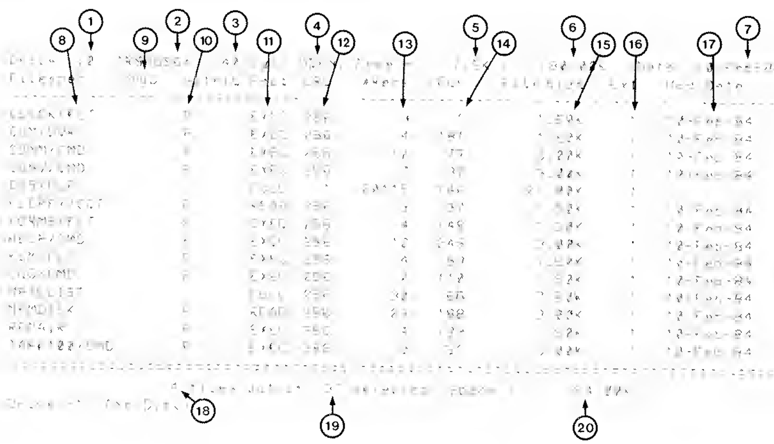
Specifying parameters allows you to select which filenames DIR prints on the screen or line printer. You must enclose parameters in parentheses. If you include more than one parameter, separate each with a comma. You cannot abbreviate the SORT and SYS parameters. The parameters are:

ALL=NO	displays the filenames for the specified drive(s). The default value is ALL=YES.
INV	displays all filenames, visible and invisible.
MOD	displays filenames modified since the last backup.
NON	enable non-stop display mode. When the directory information fills a screen, NON scrolls lines off the top of the screen.
PRT	the directory display prints on the printer and the screen. If you specify PRT, DIR assumes the NON parameter.
SYS	displays system and visible filenames.

DATE = "*date1-date2*" displays the names of files that have been modified on or after *date1* and before or on *date2*.
"*date*" displays the names of files that were modified on *date*.
"*date-*" displays the names of files modified on or after *date*.
"*-date*" displays the names of files that were modified on or before *date*.

Dates must be in the format *mm/dd/yy*.

SORT = NO does not sort the filenames. DIR assumes SORT = YES, except when you are in an application program including BASIC. Sorting is by alphabetical order.



1. Drive Number.
2. Disk Name.
3. Number of cylinders on the disk.
4. Density of the disk.
   DDEN = Double density
   SDEN = Single density
   Hard = Hard disk
5. Amount of free (unused) space on the disk. The amount is given in kilobytes (K). One K = 1024 bytes.
6. Total amount of space (used and unused) on the disk. The amount is given in kilobytes.
7. Creation date (the date the disk was formatted or was the destination in a mirror-image backup).

8. Filespec (filename and extension) assigned at creation. The filespecs are sorted alphabetically unless you specify SORT=NO in the DIR command.
9. Modification Status. A plus sign (+) indicates the file has been modified since it was last backed up.
10. File's attributes.

I	=	Invisible file.
P	=	File has an owner password.
S	=	System file.
C	=	File was created with the CREATE command or was pre-allocated by an application program.
*	=	File is a Partition Data Set (PDS) file. See the *Model 4/4P Technical Reference Manual* for additional information.
?	=	File is open. See the RESET command.

11. Protection level (the level of access assigned to the user password). See the ATTRIB command for a list of levels.
12. Length (in bytes) of the logical records in the file. The LRL (logical record length) can be any number in the range 1 to 256.
13. Number of logical records in the file.
14. End of File, or EOF (the number of the last byte in the last sector of the file).
15. Amount of space (in K) that the file takes.
16. Number of extents (non-continuous blocks of space) used to store the file. The higher the number, the more fragmented the file.
17. Modification date (the date the file was created or last written to).
18. Number of specified files (files on the disk that match the parameters you specify with DIR).
19. Total number of files on the disk.
20. Amount of space (in K) used by the specified files.

Examples

```
DIR (ENTER)
```

displays the filenames of all visible files on all enabled drives.

```
DIR :1- (ENTER)
```

displays the visible filenames on enabled drives that are equal to or greater than one.

```
DIR (INV,SYS) (ENTER)
```

displays all filenames (visible, invisible, and system) on all enabled drives.

```
DIR 0 (PRT,MOD) (ENTER)
```

prints and displays the visible filenames on Drive 0 that have been modified since the last backup. When you specify the PRT parameter, DIR assumes the NON parameter and displays the filenames on the screen without pausing. You can press (SHIFT) and (@) to pause the display. Press any key to continue.

```
DIR 1 (DATE="04/16/84-") (ENTER)
```

displays the visible filenames on Drive 1 modified on or after April 16, 1984.

```
DIR -1 (DATE="07/20/84") (ENTER)
```

displays the visible filenames on Drives 0 and 1 modified on July 20, 1984.

```
DIR /CMD:0 (ENTER)
```

displays the visible filenames on Drive 0 that have the extension /CMD.

```
DIR -/CMD:0 (ENTER)
```

displays the visible filenames on Drive 0 that do not have the extension /CMD.

Error Conditions

If you specify a drive number that does not exist or that is not enabled, DIR issues an "Illegal drive number" error message.

If you specify a range of drive numbers, each with a colon, DIR assumes that the second colon is a drive number and displays an "Illegal drive number" error message. Try the command again and only include the colon for the first drive number.

If you specify an enabled drive number that does not contain a formatted disk, DIR displays the message "[No Disk]".

DO

	Command
DO [*control character*] *filespec* [(*parameters*)] [;]	

Compiles and executes a DO file.

You can use DO to run a file of commands each time TRSDOS starts up.

A DO file is a user created Job Control Language (JCL) file that contains one or more library commands. TRSDOS executes the commands as if you had typed them in from the keyboard.

In addition to executing TRSDOS commands, you can load and execute user programs from a DO file.

You can create a DO file with the BUILD command. Command lines in this file can include library commands or filespecs. See Appendix A/ Job Control Language for more information on JCL files.

The *control characters* are:

> $ compiles your DO file without actually executing the commands.
> = executes your DO file without compiling it.
> * reruns the last DO command that was compiled.

When you specify a control character, you must leave a space between DO and the character or TRSDOS ignores the character.

The *parameters* are:

> @*label* lets you create JCL files with multiple entry points (an entry point is the place where processing begins). A label consists of the @ symbol followed by one to eight alphanumeric characters.
> parm[=*value*] lets you pass *value* to *filespec* during execution.

When you specify the @*label* parameter, *filespec* does not execute until the label is reached. Execution continues until it reaches the next label or the end of the JCL file.

The @*label* parameter, by building many different functions into one file, reduces the number of individual files on the disk (conserving space in the directory).

Use the semicolon (;) parameter when you need to specify a command line longer than 79 characters.

When a DO command line exceeds 79 characters:

1. Enclose as many parameters as will fit on one line in parentheses. Close the parentheses, insert a (;), and press (ENTER).

2. When a question mark appears on the screen, enter the remaining parameters (enclosed in parentheses).

Examples

```
DO DRIVE/JCL (ENTER)
```

compiles and executes the file named DRIVE/JCL.

```
DO = DRIVE/JCL (ENTER)
```

executes the file named DRIVE/JCL without compiling it.

```
DO $ DRIVE (ENTER)
```

compiles the file named DRIVE/JCL without executing it. Since you did not specify an extension to DRIVE, it defaulted to JCL. You can LIST the SYSTEM/JCL file to see if the JCL compiled properly.

```
DO MY/JCL (@THIRD) (ENTER)
```

compiles and executes the program named MY/JCL. All instructions in the program are ignored up to the label (@THIRD). Compilation begins at the line following the label and continues until the next label or the end of the file is reached.

```
DO * (ENTER)
```

executes SYSTEM/JCL, which contains the last DO file that was compiled.

```
DO TEST/NEW:2 (D=5,E=6) (ENTER)
```

compiles and executes the file TEST/NEW on Drive 2. The variable parameters D=5 and E=6 are passed as needed during compilation.

Error Conditions

If TRSDOS encounters an error while processing a DO command, TRSDOS does not execute the DO command.

If a JCL line is longer than 70 characters, TRSDOS displays a "Line too long" error message.

If you specify a value for token that is more than 32 characters long, TRSDOS displays a "Symbol string too long" error message.

If you specify a label that DO cannot find in the JCL file, TRSDOS displays a "Procedure not found" error message.

If you specify more than one label in a command line, TRSDOS displays a "Too many Proc labels" error message. Labels specify a point where processing begins and processing can only begin at one point.

If the diskette is not in the drive or is write-protected, TRSDOS displays a "Can't create SYSTEM/JCL file" error message.

If you assign two values to the same token in a JCL command, TRSDOS displays a "Multiply defined" error message.

If you specify more than 10 //INCLUDE statements in a JCL procedure, TRSDOS displays a "Too many nested INCLUDEs" error message.

When you specify the * *control character*, a "File not in directory" error occurs if there is no previously compiled DO file to rerun.

When you specify the $ *control character*, the system compiles the JCL file and informs you of any errors that occur. This lets you see if the file compiles properly before you actually execute it. When you compile a JCL file, a disk in your system must be write-enabled, so the system can write the compiled information to a file named SYSTEM/JCL.

When you use the = character, you cannot use some of the JCL features. See the JCL section of this manual.

Sample Uses

Suppose you want to set up the following TRSDOS functions to execute by typing one command:

```
FORMS (MARGIN=8)
TIME (CLOCK=ON)
```

Use BUILD to create such a file. If you called the file BEGIN, then use the command:

```
DO BEGIN (ENTER)
```

to perform the commands.

DUMP

Advanced Programmer's Command
DUMP *filespec* (*parameters*)

Copies an area of memory to a disk file named *filespec*.

You can use DUMP to store a machine-language program from memory to a file.

DUMP can produce a program or a core ASCII file. A program produced with DUMP can then be loaded or executed at any time. (An ASCII file cannot be loaded with the LOAD command or executed with the RUN command.)

The default extension for program dumps is /LMF, and the default extension for ASCII dumps is /TXT.

You can use some or all of the following *parameters*:

START=*address* starts the dump at *address*. You must include this parameter. The address must be above 2FFF hexadecimal or 12287 decimal.

END=*address* ends the dump at *address*. You must include this parameter. END must be greater than or equal to START, and can be either a hexadecimal or decimal number.

TRA=*address* sets the address at which your program begins executing after you load it. If you omit this parameter, any subsequent run of the file will only load the program and return you to TRSDOS Ready. TRA can be either a hexadecimal or a decimal number. The TRA and ASCII parameters are mutually exclusive.

ASCII specifies that the dump is to an ASCII file. ASCII files contain program code only. No system loading information is written to *filespec*. The TRA and ASCII parameters are mutually exclusive.

ETX=*value* specifies that the character at the end of an ASCII file is equal to *value*. *value* is a hexadecimal number in the format x'nn'. When you specify ETX, you must also specify ASCII.

ETX cannot be abbreviated.

When you DUMP to an ASCII file, you create a file that has the identical file structure as a SCRIPSIT file. The system writes a special character at the end of the file which can be changed with the ETX parameter.

Examples

```
DUMP ROUTINE/CMD (START=X'7000',END=X'8000',TRA=
X'7000') (ENTER)
```

dumps the area of memory starting at hexadecimal 7000 and ending at hexadecimal 8000. This block of memory is written to a disk file

named ROUTINE/CMD. If the file already exists, it is overwritten. If it does not exist, it is created on the first available drive. The transfer address (starting address for execution) of ROUTINE/CMD is hexadecimal 7000.

```
DUMP ROUTINE/CMD (START=28672,END=32768,TRA=
28672) (ENTER)
```

is identical to the above command except that START, END, and TRA have decimal values.

```
DUMP TEST:1 (S=X'9000',E=X'BC0F') (ENTER)
```

dumps the specified block of memory to a disk file named TEST/LMF on Drive 1. Since you did not specify a file extension to TEST, it defaulted to /LMF. Also, since you did not specify a transfer address, it is written to the file as a return to TRSDOS Ready.

```
DUMP WORD/IMG:0 (S=X'7000',E=X'A000', ASCII)
(ENTER)
```

dumps the specified block of memory to a disk file named WORD/IMG on Drive 0. Since the ASCII parameter is specified, an ASCII file is created.

```
DUMP WORD (S=X'7050',E=X'A000', ETX=X'FF', ASCII)
(ENTER)
```

dumps the specified block of memory to a disk file named WORD/TXT. An ASCII file is created, and the special character at the end of the text (end of text marker) is written as hexadecimal FF. Since you did not specify an extension for WORD, it defaulted to /TXT.

If you specify a START *address* that is less than END *address*, TRSDOS displays a "START or END error" message.

FILTER

> **Advanced Programmer's Command**
> **FILTER *devspec* [USING] *phantom devspec***

Connects a filter program to *devspec* which modifies or "filters" data as it is read from or written to *devspec*.

A filter is a program that controls the flow of data to or from a device or file. You can use a filter to change data as it passes to or from *devspec*. You can apply more than one filter to a device.

devspec is any valid, active TRSDOS device. *phantom devspec* is the name of a device which is connected to the filter program established in memory with the SET command.

You can apply as many filter programs to *devspec* as you want to. If there is not any more space in memory for the filter connection, the error message "No device space available" appears.

See the SET command for more information on FILTER.

Example

Normally, the CLICK filter (CLICK/FLT) generates a click whenever any key is pressed. By using the CLICK's CHAR=*number* parameter and the FILTER command, you can cause CLICK to generate a beep whenever an ASCII 7 (BEL) is sent to the display.

To do so, first use SET to install the CLICK filter. Then, use FILTER to connect CLICK/FLT to the display. Type:

```
SET *BP CLICK/FLT (CHAR=7) (ENTER)
FILTER *DO *BP (ENTER)
```

Now, if you press (CTRL) G or if a program sends an ASCII 7 to the display, CLICK generates a beep.

Sample Use

You can use a filter to control a printer working with non-standard size paper (see Appendices I and K, and the FORMS library command).

Error Conditions

If you specify a filter device that is already in use, TRSDOS displays a "FILTER module in use" error message. TRSDOS also displays this message if you attempt to link a filter device to more than one device.

If there is not enough memory space for the filter connection, TRSDOS displays a "No device space available" error message. Remove unused devices and try the command again.

FORMAT

FORMAT [:***drive*** [**(*parameters*)**]] **Utility**

Prepares a blank or old disk for use by defining the tracks and sectors and writing system information onto it. (For more information, see "Diskette Organization" in the *Model 4/4P Technical Reference Manual*, Cat. No. 26-2110.)

You can use FORMAT to organize a disk so you can store information on it.

drive specifies the drive in which the blank or old disk is to be formatted. If you omit the *drive*, TRSDOS prompts you for it.

The *parameters* are:

ABS overwrites any existing data without prompting. The ABS parameter is used primarily when you execute a FORMAT from a JCL file. See the JCL section for more information.
NAME="*disk name*" assigns a name to the disk being formatted.
MPW="*password*" assigns the master password to the disk. The master password allows limited access to all user files.
SDEN specifies the density of the disk as single.
DDEN specifies the density of the disk as double.
SIDES=*number* specifies the number of sides of a diskette that are to be formatted. *number* can be 1 or 2. If you omit *number*, 1 is assumed. You must have double-sided drives and media to specify SIDES=2.
CYL=*number* specifies the number of cylinders (tracks) for the disk. *number* can be 35 to 96.
QUERY=NO turns off the prompts for density, number of cylinders, name, and password.
DIR=*number* specifies the cylinder on which to put the directory. *number* must be less than the value you specify with the CYL parameter and greater than one. If you specify an invalid value or omit the DIR parameter, FORMAT assumes the center cylinder of the drive. On a 40-cylinder floppy diskette, FORMAT assumes 20.
SYSTEM recreates the directory file on a hard disk. TRSDOS assumes that you have previously formatted the hard disk. If the hard disk contains any files, the files are not accessible after you use FORMAT.

QUERY is the only parameter that can be abbreviated.

When to Format

To prepare a new disk. Before you can use a new disk, you must format it. After formatting, record the disk name, date of creation, and password. Store this information in a safe place. It helps you estimate how long a diskette has been in use. And, if you forget the master password, it ensures continued access.

To erase all data from a disk. To "start over" with a disk, you can reformat it. This erases all old information and locks out all flawed sectors which have developed. It puts the system information back on the disk and leaves the "good" sectors available for information storage.

The Format Prompts

If you specify the drive number, disk name, or master password in the command line, you are not prompted for them.

If you specify the number of cylinders, the density, or the number of sides, FORMAT does not prompt for the other two. It assumes the defaults.

If you enter a FORMAT command without specifying any parameters, you are prompted for them in the following order:

```
Which drive is to be used?
```

Enter the number of the drive you are formatting in.

```
Diskette name?
```

Enter any name with up to eight alphanumeric characters. The first character must be a letter. Press (ENTER) and the disk name defaults to DATADISK.

```
Master password?
```

Enter any password with up to eight alphanumeric characters. The first character must be a letter. Pressing (ENTER) causes the master password to default to PASSWORD.

The remaining prompts concern the type of diskette you are using. Press (ENTER) in answer to each of them if you are using standard Tandy diskettes.

```
Single or Double density <S,D>?
```

Enter (S) for single density or (D) for double density. Press (ENTER) and the value defaults to double density.

```
Number of sides <1,2>?
```

Enter (1) to format a single-sided diskette or (2) to format a double-sided diskette. You must have double-sided disk drives and media to specify two sides. Pressing (ENTER) causes the diskette to be formatted single-sided.

```
Number of cylinders?
```

Enter any number from 35 to 40 on TRS-80 hardware. Pressing (ENTER) causes the system to default to the value set with the SYSTEM (CYL=) command. TRSDOS 6 is distributed with (CYL=40) already set.

If you are formatting a disk in Drive 0 or the destination drive is not ready, the following message is displayed:

```
Load destination diskette <ENTER>
```

Insert the destination diskette and press (ENTER) to continue, or press (BREAK) to return to TRSDOS Ready.

It is important that you do not remove the system disk and insert the disk to be formatted until this prompt appears.

After you enter a format command and before the actual formatting begins, the system checks the destination diskette to see if it is already formatted.

If the disk is formatted and its MPW is PASSWORD, the following message appears:

```
Disk contains data -- Name=disk name
  Date=mm/dd/yy
Are you sure you want to format it?
```

Press (N) to abort the FORMAT or (Y) to continue. If you specified the ABS parameter in the command line, you see the DISK CONTAINS DATA message, but you are not prompted to abort the format.

If the disk is formatted and the master password of the destination disk is not PASSWORD, the following message appears:

```
Disk contains data -- Name=disk name
  Date=mm/dd/yy
Enter its Master Password or <BREAK> to abort
```

Press (BREAK) to abort the format or enter the master password to continue.

If the disk contains an incomplete or non-standard format, one of the following messages may appear in place of NAME=*disk name*:

```
Unreadable directory
Non-standard format
Non-initialized directory
```

When the format begins, you see the cylinder numbers appear as the necessary information is written to them. After all cylinders are written, FORMAT verifies that the proper information is actually on each cylinder.

If the verify procedure detects an error, an asterisk and the cylinder number are shown on the screen. That cylinder is locked out, so that no files can be written to the defective area. Use the FREE library command to see the locked out cylinders on a diskette.

During parts of the format operations, the system real time clock is turned off.

After formatting is complete, you are prompted to put the system disk back in Drive 0 with the message:

```
Load SYSTEM diskette <ENTER>
```

The format is now complete.

Examples

```
FORMAT (ENTER)
```

prompts you for the drive number, the diskette name, the master password, the density, the number of sides, and number of cylinders, and checks to see if the destination disk is already formatted.

```
FORMAT :1 (NAME="DATA3",MPW="SECRET") (ENTER)
```

prompts you for the DEN, SIDES, and CYL parameters, and checks to see if the disk in Drive 1 is already formatted. The disk in Drive 1 is assigned the name DATA3 and the master password SECRET.

```
FORMAT :0 (NAME="FILES", MPW="FILE01", Q=N) (ENTER)
```

displays the message:

```
Load destination diskette <ENTER>
```

When you insert the destination disk in Drive 0, the system checks to see if the disk is already formatted. When the message:

```
Load SYSTEM diskette <ENTER>
```

appears, insert the system disk in Drive 0 and the format is completed.

```
FORMAT :1 (QUERY=NO,ABS) (ENTER)
```

formats the disk in Drive 1 (with the default options) even if the disk already contains data. Because you specified the ABS parameter, you don't have the opportunity to abort the FORMAT (if the disk is already formatted and its master password is PASSWORD).

Error Conditions

FORMAT builds a track image of the diskette in memory. If there is not enough memory available to build this track, TRSDOS displays an "Insufficient memory for specified format" error message. TRSDOS usually displays this message when HIGH$ is set very low and the FORMAT program requires memory above HIGH$. Reset HIGH$ and try the FORMAT command again.

If you attempt to use the SYSTEM parameter with a diskette, TRSDOS displays a "Cannot 'SYSTEM' a floppy" error message. The SYSTEM parameter is only valid with hard disks.

If you specify an invalid drive number, the error message "Illegal drive number" appears.

If there is write-protect tab on the disk or the drive is protected by the SYSTEM (DRIVE=,WP) command, the error message "Disk write protected" appears.

If the error message "Load destination diskette" appears after you insert the destination diskette:

1. Be sure the disk drive door is closed.
2. Make sure that the diskette is inserted correctly.
3. If the drive is an external drive, make sure the drive is connected and plugged into the outlet.
4. Try formatting the disk in a different drive.

If the message "Formatting complete" does not appear, repeat the format. If the message still does not appear, it indicates a flawed disk. Try another disk.

FORMS [(*parameters*)]	**Command**

Sets up forms filter parameters.

You can use FORMS to print a form larger or smaller than a standard-size page.

The *parameters* are:

DEFAULT returns all parameters to their start-up values.
ADDLF issues a linefeed after every carriage return.
CHARS = *number* sets the number of characters per printed line. *number* is 1 - 255.
FFHARD issues a form feed (Top of Form) character (ASCII code 12) instead of a series of linefeeds.
INDENT = *number* sets the number of spaces a line is to be indented if the line length exceeds CHARS. The default value for *number* is 0.
LINES = *number* sets the number of lines to be printed per page. The default value for *number* is 66.
MARGIN = *number* sets the left margin.
PAGE = *number* sets the physical page size as *number* of lines. The default value for *number* is 66.
QUERY prompts you for each parameter.
TAB specifies that tab characters are to be translated into the appropriate number of spaces.
XLATE = *X'aabb'* specifies a one-character translation to be performed by the filter.
aa is the character in hex format to be translated.
bb is the character in hex format *aa* is translated to.

To determine the parameters to set for:

page size	multiply form length in inches by the number of lines per inch.
lines per page	determine the number of blank lines at the bottom of every page. The default number of blank lines is 0. If LINES = PAGE, then text can be written on every line of each page. LINES cannot exceed PAGE.
characters per line	multiply form width in inches by the number of characters per inch (10 or 12). Use CHARS to set the maximum number of printable characters per line. If a line is greater than CHARS, then TRSDOS automatically breaks the line at the maximum length, and continues printing at the next line. The line is indented if you have specified INDENT.

Examples

Be sure that you have SET *FF to its filter program FORMS/FLT and you have FILTERed the printer to *FF with the commands:

```
SET *FF TO FORMS/FLT (ENTER)
FILTER *PR *FF (ENTER)
```

```
FORMS (ENTER)
```

displays the current parameter values.

```
FORMS (CHARS=80,INDENT=6,PAGE=51,LINES=45,FFHARD)
(ENTER)
```

allows a maximum of 80 characters per printed line. If a line contains more than 80 characters, the excess is printed on the next line and indented 6 spaces. The physical page size is set to 51 lines, and 45 lines can be printed on a page.

FFHARD allows the printer to manage top-of-form. Most printers that support top-of-form have a control that must be set to the length of the paper for FFHARD to function properly. The default for this control is usually 66 lines per page. Not all printers support top-of-form. Check the owner's manual for the printer you are using for information on top-of-form control.

FFHARD is faster than sending a series of line feeds. If you are using the system spooler, FFHARD requires less space in memory and on disk. See the SPOOL command.

```
FORMS (MARGIN=10,CHARS=80,INDENT=16) (ENTER)
```

causes all lines to start 10 spaces in from the normal left-hand starting position. Any line longer than 80 characters is indented 16 spaces (6 spaces after the margin) when wrapped around, so it is printed starting at position 16.

```
FORMS (TAB,ADDLF) (ENTER)
```

specifies that tab characters are to be translated into the appropriate number of spaces. Also, a linefeed is sent to the printer every time a carriage return is sent.

```
FORMS (XLATE=X'2A2E') (ENTER)
```

translates all hexadecimal 2A characters (asterisks) to hexadecimal 2E characters (periods).

Sample Uses

Suppose you have a payroll program that contains all of your employees' payroll information, and that prints checks of the size 4" × 7".

To instruct your computer to print a form 4" × 7", issue the following commands:

```
SET *FF TO FORMS/FLT (ENTER)
FILTER *PR *FF (ENTER)
FORMS (CHARS=55,LINES=20,PAGE=24) (ENTER)
```

Now when you run your payroll program, you can print the checks on the proper size form.

FREE

FREE [:*drive*] [(*parameter*)]	**Command**

Lists the amount of space that is free (available for use) and the number of files on each drive, if no *drive* is specified. If *drive* is specified, displays a free-space map of the disk in that drive.

You can use FREE to see how many files are on a disk. You can also use FREE to see a table containing information about each disk in your computer.

The parameter is:

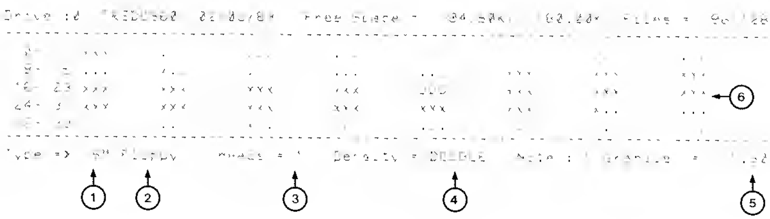
PRT sends output to the printer.

FREE displays free-space information about each enabled disk in the following format:

```
        1      2          3                             4          5                 6   7
Drive :0  TRSDOS6X  02/02/83   Free Space =   94.50K/  180.00K   Files =   96/128
Drive :1  TRSDOS6X  02/17/83   Free Space =   91.50K/  180.00K   Files =   97/128
```

1. Drive number.
2. Disk name.
3. Creation date (the date the disk was formatted or was the destination in a mirror-image backup).
4. Amount of free (unused) space on the disk. The amount is given in kilobytes (K). One K = 1024 bytes.
5. Total amount of space (used and unused) on the disk. The amount is given in kilobytes.
6. Number of free (unused) directory slots. A file can take one or more directory slots.
7. Total number of directory slots on the disk. Some of these are used by the operating system.

FREE displays a free space map of the specified disk in the format shown below.



1. Disk size.
2. Drive type: floppy or hard.
3. Number of *heads* (surfaces on the disk) that contain data for this logical drive.
4. Density of the disk: SINGLE or DOUBLE.
5. Amount of space (in kilobytes) in each *granule* on the disk. One kilobyte, or K, equals 1024 bytes.
6. Space allocation map, which shows the organization of data on the disk.

   The numbers in the left column represent the *cylinders* on the disk. Cylinders are divided into granules, or *grans*. The status of each gran is represented by a character on the map, as follows:

   . = Available for use.
   * = Locked out (unavailable because it is flawed).
   X = Currently used by system or user files.
   D = Currently used by the system's directory files.

On hard disks, X indicates that a granule is in use or is flawed. The asterisk (*) is not used on hard disks.

Examples

```
FREE (ENTER)
```

displays free space information about each enabled disk.

```
FREE :0 (PRT) (ENTER)
```

displays a free space map of the disk in Drive 0. The map is also sent to the printer because you specified the PRT parameter.

HELP

HELP [***filespec*** [*][***keyword***] [(***parameter***)]]	**Utility**

Displays information about TRSDOS keywords.

filespec is the file specification for the HELP data file you want to access. Your TRSDOS system diskette contains one HELP data file, DOS/HLP. The extension /HLP is optional.

keyword is the TRSDOS command, filter, or driver about which you seek information.

parameter specifies additional information for the HELP display. You must enclose *parameters* in parentheses and separate them with commas. *parameter* may be any of the following:

P prints the information on the printer as well as the screen.
V cancels video restoration.
R cancels reverse video when displaying information on the screen.
S displays only commands that match the command partspec you indicate with *keyword*.

If you include *keyword* or *parameter* you must include *filespec*.

Some applications programs may have HELP data files for information about the application. You can use an asterisk in the HELP command line to indicate a global search for the keyword in all HELP data files. The asterisk must immediately precede *keyword*.

If you type HELP at the TRSDOS Ready prompt, HELP displays a reference menu. The menu shows possible syntax combinations you can use with the HELP command from the TRSDOS Ready prompt.

The menu also lists the HELP data files currently on-line (available). The HELP data files are called categories in the reference menu. The menu displays the message:

```
Enter category or press <ENTER> to exit:
```

In response to this prompt, you can enter DOS or DOS *keyword*. You can also include parameters.

If you press (ENTER) or (BREAK), HELP returns to the TRSDOS Ready prompt. If you enter an invalid category or omit category, HELP redisplays the reference menu.

When you enter DOS, HELP displays a list of all keywords available in DOS, followed by the message:

```
Enter keyword or press <BREAK> to exit:
```

Enter the keyword that you want information for. If you press **(BREAK)**, HELP returns to the TRSDOS Ready prompt. If you press **(ENTER)**, HELP redisplays the list of keywords.

If you enter DOS keyword, HELP displays the information available in DOS for that keyword and the message:

```
Enter keyword or press <BREAK> to exit:
```

If the information for a keyword requires more than one screen, HELP displays a message that you should press any key to see additional information.

If you press **(BREAK)**, HELP returns to the TRSDOS Ready prompt. If you enter another keyword, HELP displays the information available in DOS about that keyword. If you enter an invalid keyword or press **(ENTER)**, HELP redisplays the list of keywords.

If an application program contains a HELP data file, you can enter **keyword* from the reference menu to perform a global search of all HELP data files for category. When performing a global search, HELP searches all available data files for *keyword*. While searching, HELP displays the message:

```
Global Search in File: filename
```

When HELP finds the keyword in a data file, HELP displays the keyword information on the screen and the message:

```
Press <BREAK> to exit or <ENTER> to continue
Global Scan
```

If you press **(ENTER)**, HELP continues the search for keyword in other data files. When HELP completes the global search, HELP displays the message:

```
Global Search in File:filename
END of Global Scan

Press <ENTER> to exit:
```

When you press **(ENTER)**, HELP returns to the TRSDOS Ready prompt.

Examples

At the TRSDOS Ready prompt type:

HELP **(ENTER)**

HELP displays the reference menu.

At the TRSDOS Ready prompt type:

```
HELP DOS (ENTER)
```

HELP displays the list of all keywords in the DOS file.

At the TRSDOS Ready prompt type:

```
HELP DOS ATTRIB (ENTER)
```

HELP displays a screen of information available for the ATTRIB command and prompts you to press any key for additional information. After you press a key, HELP displays additional information and prompts you to enter another keyword or press (BREAK) to exit.

LIB

LIB	Command

Displays a listing of all system commands in Libraries A, B, and C.

You can use LIB to see a list of TRSDOS commands.

Library A contains the primary TRSDOS commands, Library B contains the secondary commands, and Library C contains the machine-dependent commands.

Example

```
LIB (ENTER)
```

displays a list of the TRSDOS library commands.

```
Library <A>
  Append  Cat     Cls     Copy    Device  Dir     Do
  Filter  Lib     Link    List    Load    Memory  Remove
  Rename  Reset   Route   Run     Set     Tof
Library <B>
  Attrib  Auto    Build   Create  Date    Debug   Dump
  Free    Purge   Time    Verify
Library <C>
  Forms   Setcom  Setki   Spool   Sysgen  System
```

Technical Information

Library A is located in the SYS6/SYS system module, Library B is located in the SYS7/SYS system module, and Library C is located in the SYS8/SYS system module. You can remove any of the three system modules if you will not be using their commands. (Use the PURGE or REMOVE library commands to delete system modules.)

Advanced Programmer's Command
LINK *devspec1* [TO] *devspec2*

Links together two logical devices; both must be enabled in the system.

You can use LINK to get a printout of the data displayed on your video display. You can also use LINK to write data displayed on the screen to a disk file.

To "unlink" the devices, use the RESET command.

Be careful if you make several links to the same device. You could create an endless loop and hang up the system.

Examples

```
LINK *DO *PR (ENTER)
```

links the video display (*DO) to the line printer (*PR). All output sent to the display (*devspec1*) is also sent to the line printer (*devspec2*).

NOTE: Although all output to the video display is also sent to the printer, any output sent individually to the printer (such as an LPRINT from BASIC) is *not* sent to the video display. This is because the order of the devices in the link command line is important. Once linked, any information sent to *devspec1* is also sent to *devspec2*, and any information requested from *devspec1* can also be supplied by *devspec2*. However, information sent to *devspec2* is not sent to *devspec1*, nor can information requested from *devspec2* be supplied by *devspec1*.

```
LINK *PR *DO (ENTER)
```

links the line printer to the video display. All output sent to the printer (*devspec1*) is also sent to the video display (*devspec2*).

Linking a Device To a File

It is not possible to directly LINK a device to a file. To link a device to a file, follow this procedure:

- Use the ROUTE library command to create a "phantom" device and route it to the file.
- Link the device to the phantom device using the LINK library command.

NOTE: Do not use the SYSGEN library command if you currently have a device linked to a file. The linked file is shown as open every time you power up or reset the system. You can overwrite other files very easily if you switch disks with the linked file open.

The following example shows how to link your line printer to the disk file PRINT/TXT on Drive 0 using a phantom device.

```
ROUTE *DU TO PRINT/TXT:0 (ENTER)
```

creates the phantom device *DU and routes it to the disk file PRINT/TXT on Drive 0. If PRINT/TXT does not exist, it is created. If it already exists, data sent to the file is appended onto its end.

```
LINK *PR *DU (ENTER)
```

links the printer to *DU, which in turn is routed to PRINT/TXT. All output sent to the line printer is also sent to *DU (that is, written to PRINT/TXT).

NOTE: PRINT/TXT remains open until you issue a RESET *DU command. To break the link between the printer and PRINT/TXT without closing the file, use the RESET *PR command. See the ROUTE and RESET library commands; the "Using the Device-Related Commands" section, and Appendices I and K.

Sample Use

Suppose you want your computer to be accessed by another computer or terminal. If your computer is at the office you can use a remote terminal as the keyboard and display of your computer. First, set the Communications Line device (*CL) and use SETCOM to specify WORD=8 and PARITY=NO with the commands:

```
SET *CL TO COM/DVR (ENTER)
SETCOM (WORD=8,PARITY=NO) (ENTER)
```

then issue commands:

```
LINK *DO *CL
LINK *KI *CL
```

to link the video display and keyboard to the RS-232-C interface. This lets your computer act as a "host" and be accessed by a remote terminal via the RS-232-C hardware.

While these links are in place, anything typed on your computer's keyboard or the remote terminal keyboard is treated as if it originated at your keyboard. Text displayed on your computer's screen is transmitted to the remote terminal.

Note: Some programs display data using a direct access method. This data is not displayed on the remote terminal. For more information about direct access, see the *Model 4/4P Technical Reference Manual* and the @VDCTL SVC.

LIST *filespec* [(*parameters*)]	**Command**

Lists the contents of *filespec*.

You can use LIST to see the contents of a file on a disk.

The parameters are:

ASCII8 displays the graphic characters and special characters in a file, along with the text.
NUM numbers the lines in ASCII text files.
HEX specifies hexadecimal output format. When you specify the HEX parameter, NUM and LINE are ignored.
TAB = *number* specifies that tab stops are to be placed every *number* of spaces apart for ASCII text files. Each tab character (ASCII 9) encountered causes a jump to the next tab stop. The default value for *number* is 8.
PRT directs output to the printer.
LINE = *number* sets the starting line to *number*. If you omit the LINE = parameter, TRSDOS uses 1. This parameter works only with ASCII files.
REC = *number* sets the starting record number to *number*. If you omit the REC = parameter, TRSDOS uses 0. The REC = parameter is used only with the HEX parameter.
LRL = *number* sets the logical record length to be used to display a file with a record length of *number*. If you omit the LRL = parameter, TRSDOS uses the logical record length of the file. The LRL = parameter is used only with the HEX parameter.

LINE cannot be abbreviated, and the abbreviation for ASCII8 is A8.

If you omit a file extension with LIST command, TRSDOS looks for the filename with the extension /TXT. If TRSDOS cannot find the filename with the /TXT extension, it looks for *filename.* If you specify *filename* with the extension / you can eliminate the search for *filename*/TXT.

Press (SHIFT)(@) to pause a list. Press any key to continue. Press (BREAK) to abort the list.

When you use the HEX parameter, *filespec* is listed in the following format:

1 2 3 4

[illegible]

1. Current logical record of the file in hex notation, starting with record 0.
2. Offset from the first byte of the current logical record (in hex notation).
3. Hex representation of the byte listed.
4. ASCII representation of the byte. A period is used for all non-ASCII bytes.

Examples

```
LIST TESTFILE:0 (ENTER)
```

searches Drive 0 for TESTFILE/TXT. If not found, it searches for TESTFILE.

```
LIST MONITOR/CMD (HEX,LRL=8)
```

lists MONITOR/CMD in the hexadecimal mode using an LRL of 8.

```
LIST REPLY/TXT (NUM,TAB=10,P) (ENTER)
```

prints a listing of REPLY/TXT on the printer, numbering each line that is printed. Lines are numbered beginning with 00001. Any tab character encountered causes a jump to the next tab position (every 10th column).

```
LIST TESTFILE/OBJ (HEX,REC=5) (ENTER)
```

list TESTFILE/OBJ in the hex mode, beginning with record 5.

Sample Use

Suppose you create a JCL file that contains a series of commands you often execute in a given order. To review the contents of the file, type:

```
LIST STARTUP/JCL (ENTER)
```

LOAD

Advanced Programmer's Command

LOAD [(*parameter*)] ***filespec***

Loads a machine-language program file (without executing it) and then returns to TRSDOS Ready.

You can use LOAD to pre-load assembly language routines that programs written in a language such as BASIC can call.

The parameter is:

X loads a file from a non-system disk.

The file must be in load module format. Do not use it to load BASIC program files. The default file extension for the LOAD command is /CMD.

Programs to be loaded must reside at or above the address X'3000'.

Examples

```
LOAD STATUS/CMD (ENTER)
```

loads the file STATUS/CMD into memory.

```
LOAD (X) PROGRAM/CIM (ENTER)
```

loads PROGRAM/CIM from a non-system disk. The system prompts you to insert the disk with the desired file on it with the message:

```
Insert SOURCE disk <ENTER>
```

After the file is loaded, you are prompted to put the system disk back in Drive 0 with the message:

```
Insert SYSTEM disk <ENTER>
```

The load is now complete.

Sample Use

Often several program modules must be loaded into memory for use by a master program. For example, suppose PAYROLL/PT1 and PAYROLL/PT2 are modules, and MENU/CMD is the master program. Then you could use the commands:

```
LOAD PAYROLL/PT1 (ENTER)
LOAD PAYROLL/PT2 (ENTER)
```

to get modules into memory, and then type: MENU to load and execute MENU.

LOG/CMD

LOG :*drive*	**Utility**

Lets you exchange one type of disk for another. LOG informs the system that it needs to re-examine the specified *drive* to determine the disk type, cylinder count, density, number of sides, and location of the directory.

:drive can be any enabled drive in your system. If you omit *drive*, the system re-examines Drive 0.

Use LOG whenever you switch diskettes in Drive 0. If you switch from TRSDOS 06.02.*xx* to TRSDOS. 06.02.*xx*, you can use LOG, instead of resetting the system. But if you switch from TRSDOS 06.*xx* to TRSDOS 06.*xx*, you **must** press RESET.

Some minimum system diskettes do not contain the LOG command. If this is the case with your Drive 0 diskette, you must reset the system, instead of using LOG.

After you enter the LOG command, if you are changing systems diskettes in Drive 0, TRSDOS displays the message:

[illegible]

When TRSDOS displays this message, remove the diskette in Drive 0. Insert the new diskette and press (ENTER). TRSDOS is now aware of the fact that Drive 0 contains a different type of system diskette.

MEMORY

MEMORY [(*parameters*)]	**Advanced Programmer's Command**

Allows you to reserve a portion of memory, display or change the current HIGH$ and LOW$, modify a memory address, or begin executing at a specified memory location. HIGH$ has to be higher than LOW$.

You can use MEMORY to find out which area of memory you can use.

The *parameters* are:

CLEAR=*value* fills memory from hex 2600 to HIGH$ with *value*. *value* in the format X'nn'. If you do not specify *value*, memory is filled with the hexadecimal value 00 (null).

HIGH=*address* resets HIGH$ to the address you specify. *address* must be less than the current value of HIGH$. When you reset HIGH$, TRSDOS inserts a high memory header into memory. TRSDOS places the header in the last 10 bytes preceding the address you specify. TRSDOS displays a message informing you that the header is in memory.

LOW=*address* resets LOW$ to the address you specify. *address* must be greater than Hexadecimal 25FF. If you omit *address*, TRSDOS displays the current value of LOW$. Subsequent MEMORY or system level commands reset LOW$ to its default value of Hexadecimal 2600.

ADD=*value* displays or modifies a byte or word of memory. *value* can also be a memory address or an alphabetic character A through Z, specifying a TRSDOS flag in the system flag table. See the *Model 4/4P Technical Reference Manual* for additional information on the status flag table. If you include the WORD or BYTE parameters, ADD modifies the address or flag that you specify with value. If you omit WORD and BYTE, ADD displays the address or flag that you specify with value.

WORD=*word* changes the contents of ADD and ADD + 1 to *word*.

BYTE=*byte* changes the contents of ADD to *byte*.

GO=*address* transfers control to *address*. If more than one parameter is specified, the GO parameter is always executed last.

address is any memory address in hexadecimal or decimal notation. *word* is any value in the range 0000 - FFFF hexadecimal or 0 - 65535 decimal. *byte* is any value in the range 00 - FF hexadecimal or 0 - 255 decimal.

Examples

```
MEMORY (ENTER)
```

displays HIGH$ (the highest unused memory location) and LOW$ (the lowest reserved memory location) in the hexadecimal X'nnnn' format.

```
MEMORY (HIGH=X'E100') (ENTER)
```

sets HIGH$ to hexadecimal memory address E100, as long as the existing HIGH$ is above X'E100'. The MEMORY command moves HIGH$ lower in memory.

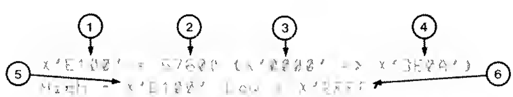
```
MEMORY (ADD=X'6500') (ENTER)
```

displays the contents of hexadecimal memory addresses 6500 and 6501 in the following format:

1 The address specified in hexadecimal notation.
2 The decimal equivalent of the address.
3 The contents of address and address + 1, in MSB-LSB format.
4 The current HIGH$ address.
5 The current LOW$ address.

```
MEMORY (ADD=X'E100',WORD=X'3E0A') (ENTER)
```

modifies hexadecimal memory locations ADD (E100) and ADD+1 (E101), changing them to the value of WORD. The following display appears:



1 The address specified in hexadecimal notation.
2 The decimal equivalent of the address.
3 The old contents of address and address + 1, in MSB-LSB format.
4 The new contents of address and address + 1, in MSB-LSB format.
5 The current HIGH$ address.
6 The current LOW$ address.

```
MEMORY (ADD=X'E100',BYTE=X'C9') (ENTER)
```

changes the BYTE of memory at hexadecimal address E100 to hexadecimal C9. The display after executing this command is:

```
X'E100' = 57600 (X'00' => X'C9')
High = X'E100'  Low = X'2FFF'
```

The display is identical to the last example, except that the command modified a BYTE instead of a WORD.

```
MEMORY (GO=X'E100') (ENTER)
```

transfers control to hexadecimal memory address E100.

Error Conditions

If you specify an address for LOW that is equal to or greater than the address for HIGH, TRSDOS displays a "Range error" message. TRSDOS also displays this message if you attempt to set HIGH greater than the current value of HIGH or if you attempt to set LOW less than X'2600'.

Some applications programs set a bit in CFLAG$ that does not allow you to alter HIGH$ or LOW$. If you attempt to alter these values when this bit is set, TRSDOS displays a "No memory space available" error message. The application program must reset the bit in CFLAG$. See the *Model 4/4P Technical Reference Manual* for additional information on CFLAG$.

Method A

Advanced Programmer's Utility
PATCH *filespec* (*patch commands*)

Method B

Advanced Programmer's Utility
PATCH *filespec1* USING *filespec2* [(*parameters*)]

Lets you make minor corrections in any disk file by (1) typing in the patch code directly from the command line (Method A), or (2) creating an ASCII file containing patch information (Method B).

You can use PATCH to make minor changes in your own machine-language programs. You need not change the source code, reassemble it, and recreate the file. You can use PATCH to make minor replacement changes in data files, also.

filespec1 is the file to be changed and /CMD is its default extension. *filespec2* contains the patch commands. *filespec2* can contain only ASCII characters and /FIX is its default extension.

The *patch commands* are:

address=*value* identifies the PATCH as a patch by "memory load location." It changes the contents of memory beginning with *address* to *value*.

D*record,byte*=*value* identifies the PATCH as a "direct modify patch." *record* tells which record contains the data to be changed. It is a hexadecimal number from 00 to FF. *byte* specifies the position of the first byte to be changed. It is a hexadecimal number from 00 to FF.

F*record,byte*=*value* lets you make sure that a patch is applied to the correct place in memory, when used in conjunction with the D *patch command*. F*record,byte* follows D*record,byte*. If the location specified with the D *patch command* does not contain the data specified with F*record,byte* , the PATCH aborts. F*record,byte* is also used with the REMOVE parameter to remove a patch and replace it with the original data.

L*code* identifies the PATCH as a "library mode patch." The PATCH applies to either the SYS6/SYS, SYS7/SYS, or SYS8/SYS library command module. *code* is the binary coded location in the format *nn* where the change begins.

address is a four-digit hexadecimal value in the format X'*nnnn*' which is the memory load address for the change.

value can be either a series of hexadecimal bytes in the format *nn nn nn...*, or a string of ASCII characters in the format "*string*."

The *parameters* are:

YANK removes the PATCH specified by *filespec2* from *filespec1*. The specified PATCH contains code in the *address* format.
REMOVE removes the PATCH specified by *filespec2* from *filespec1*. The specified PATCH contains code in the D*record,byte* format.

An **address** *patch command* changes a file by "memory load location." It adds the patch code to the end of the *filespec* and then makes the changes beginning at *address* each time the file is loaded. You can use YANK to remove the added code from *filespec*. This type of patch can be applied only to standard "load modules" or command (/CMD) files.

A ***Drecord,byte;Frecord,byte*** *patch command* changes a file by "direct modify patch." It changes a file by directly applying the patch code to the specified record and byte of *filespec*. When you BUILD a file containing *patch commands* in this format, you can REMOVE the patch.

An easy way for you to find the record and byte of *filespec* that you want to patch is to list *filespec* using the LIST library command with the HEX parameter. Remember that the first record in a file is record 0, not record 1.

A direct modify patch can be specified on a command file, as well as from a file. The format for patching on a command line is:

```
PATCH filespec (Drecord,byte=value:
Frecord,byte=value
```

Notice the colon. On a command line, use a colon to place more than one item on a line. In a patch file, use a semicolon. Also, notice that you must include the Find portion of the patch on the line. If your patch is long, break it into several command lines.

Lcode *patch command* patches are supplied by Tandy for you to implement changes to TRSDOS. You have to BUILD a patch file to apply this type of PATCH.

Examples

These examples are used to show the syntax and development of the PATCH command, so do not execute them.

```
PATCH MONITOR/CMD (X'E100'=C3 66 00 CD 03 40)
(ENTER)
```

patches the file MONITOR/CMD by the memory load location method. The six bytes beginning at hexadecimal E100 are changed. During the PATCH operation, the following message is displayed:

[illegible]

When the operation is completed, this message appears:

```
Patch function completed
x patch lines installed
```

x is the number of lines of patch code that were installed.

Since there is no filespec used for the patch code, the name CLP (Command Line Patch) is assigned to the patch code. You can use this name to later YANK the patch from MONITOR/CMD.

```
PATCH PROFILE/CMD (D02,45=C3 64 66 69 76:F02,45
   =C3 11 45 FE 45)
```

causes the patch utility to examine record 2, byte X'45'. If the sequence C3 11 45 FE 45 is found at that location, the sequence C3 64 66 69 76 is written in its place. If the find string is not found, no bytes are altered.

Error Conditions

If you omit *filespec* in a PATCH command, TRSDOS displays a "PROGRAM file name required" error message. Include *filespec* and try the command again.

If you misspell *filespec*, TRSDOS displays a "File not in directory" error message. Check your spelling and try the command again.

If you attempt to YANK an X patch that is not in the file, TRSDOS displays a "Can't yank patch not in load file" error message. TRSDOS also displays this message if you attempt to YANK a patch that was applied to the same file more than once and has been previously yanked.

If you specify an incorrect overlay number in an L patch to SYS6, SYS7, or SYS8, TRSDOS displays a "Library overlay not found" error message.

If there are errors in the /FIX file, TRSDOS may display any of the following error messages:

"Invalid library format"
"Patch input format error"
"Non-hex digit encountered"

If the file in an X patch is not in load file format, TRSDOS displays a "Load file format error" message.

If the /FIX file is too large, TRSDOS displays a "Fix file too big — partition it" error message. You must split the file into more than one file.

Examples

Using BUILD To Create a PATCH File

You can use the BUILD library command to create a PATCH file. (See the "Building a File" section of the BUILD library command.) A PATCH file can contain only ASCII characters.

Each line in a patch file is either a *patch command* or a comment. Comment lines begin with a period and are ignored by the patch utility. Use comments in patch files to document the changes that you make. You can append a comment onto the end of a *patch command* by using a semicolon to separate the two parts.

These examples are used to show the syntax and development of the PATCH command, so do not install them.

```
PATCH BACKUP/CMD:0 USING SPECIAL/FIX (ENTER)
```

The data in BACKUP/CMD on Drive 0 is changed to 23 3E 87 beginning at hexadecimal 6178. The data beginning at hexadecimal 61A0 is changed to FF 00 00. This is an example of a memory load location patch, and since the patch is added onto the end of BACKUP/CMD, you can use the YANK parameter to remove it.

Use the BUILD library command to create the following PATCH file named TEST/FIX:

```
.This patch modifies the SYS2 module.
D0B,49=EF CD 44 65;F0B,49=DD 3A 33 44
D0B,55=C3 00 00;F0B,55=EF 44 55
.End of patch
```

Now, type in the command line:

```
PATCH SYS2/SYS.PASSWORD USING TEST/FIX (ENTER)
```

changes the data specified in SYS2/SYS.PASSWORD to the data in TEST/FIX. Since the data beginning at record 0B, byte 49 and 55 is directly changed on disk, this is an example of a direct disk modify patch. The Find *patch commands* let you make sure you are patching the correct place in memory.

Using PATCH on a TRSDOS System File

When Tandy releases a modification to TRSDOS, you receive a printout of the exact *patch commands* that you must use to make the change.

Suppose Tandy sends you the patch information for a file named LIB1 that contains the following patch code:

```
.USE LIB1 TO PATCH SYS6/SYS.
L54
X'5208'=32 20 DE AF 00 C3 66 00
```

Use the BUILD library command to create the file LIB1, and type in the lines exactly as they appear on the printout. After you end the file, type in the command line:

```
PATCH SYS6/SYS:1 USING LIB1 (ENTER)
```

This changes the data specified in SYS6/SYS on Drive 1 to the data in LIB1/FIX. Since you did not specify an extension to LIB1, it defaulted to /FIX. This patch is in the memory load location mode. Library patches can also be done with the direct disk modify mode. To be sure that you do not patch the disk in Drive 0, specify the drive number in the filespec (such as SYS6/SYS:2) or write protect the disk in Drive 0.

PATCH lets you implement any changes to TRSDOS that may be supplied by Tandy. This way, you do not have to wait for a new release of TRSDOS.

To make a change, follow these general steps:

1. Make a backup copy of the diskette to be patched.
2. Insert the TRSDOS diskette to be changed into one of the drives. (Make sure the diskette is "write-enabled.")
3. In the TRSDOS mode, use the BUILD library command to create a PATCH file containing the patch commands specified in the information provided by Tandy.
4. Issue the appropriate PATCH command.
5. After the patch is complete, test the patched diskette in Drive 0 to see that it is operating as a TRSDOS system diskette. You have to reset the computer before you can test the diskette.

PURGE

	Command
PURGE [*partspec* ! -*partspec*]:*drive* [(*parameters*)]	

Deletes all or some files from the disk in *drive*.

If you specify *partspec*, PURGE deletes all files that match *partspec*. If you specify *partspec*, preceded by a hyphen (-), PURGE deletes all files that do not match *partspec*. If you include a drive number with *partspec*, you must include the colon (:)

If you don't know the file's password but you know the disk's master password, PURGE provides a way to delete the files. With the REMOVE command you must know the file password to delete a file. With the PURGE command, you must specify the disk's master password unless the master password is PASSWORD.

The parameters are:

QUERY=NO automatically removes files without prompting for each one.
MPW="*password*" states the disk master password
INV removes the invisible files as well as the visible files.
SYS removes the system files as well as the visible files.
DATE="*date1-date2*" deletes the files that were modified on or after *date1* and on or before *date2*.
="*date*" deletes the files that were modified on *date*.
="*date-*"deletes the files that were modified on or after *date*.
="-*date*" deletes the files that were modified on or before *date*.
Dates must be in the format *mm/dd/yy*.

Before using the PURGE command, you can see which files will be purged by executing a DIR command with the same parameters.

Once you enter the PURGE command, TRSDOS prompts you for the disk's password (if it is not PASSWORD), unless you specified it with the MPW parameter.

Then, the system displays the files one at a time. Unless you specify QUERY=NO, it prompts you to remove the file or keep it. Respond with (Y) to remove the file, (N) or (ENTER) to keep it.

NOTE: BOOT/SYS and DIR/SYS cannot be purged and do not appear during execution of any PURGE command.

Examples

```
PURGE :0 (MPW="SECRET") (ENTER)
```

purges all visible files from Drive 1. Before removing each file, the system asks if you're sure you want it to do so.

```
PURGE /BAS:1 (Q=NO) (ENTER)
```

purges all visible files with the extension /BAS from Drive 1. You are not questioned before each file is removed, as QUERY is specified as NO.

```
PURGE /$$S:2 (ENTER)
```

purges all visible files on Drive 2 whose file extension contains 3 characters and ends in the letter S.

```
PURGE -/CMD:0 (INV) (ENTER)
```

purges all non-system files from Drive 0 except those with the extension /CMD.

```
PURGE :1 (DATE="02/01/81-") (ENTER)
```

purges all visible files on Drive 1 with modify dates of February 1, 1981 or later. You are questioned before each file is removed.

Error Conditions

When you use a PURGE command in a JCL file you must specify the QUERY=NO parameter. If you omit this parameter, TRSDOS displays an "Invalid command during <DO> processing" error message.

If you specify the wrong password, TRSDOS displays an "Invalid master password" error message. If the master password is not PASSWORD, you must specify it to PURGE a file.

Sample Use

Refer to the sample in the BACKUP command section. Now that you have moved all of the new files to the disk in Drive 1, you can remove all the new files on the disk in Drive 0 by issuing the command:

```
PURGE /NEW:0 (ENTER)
```

Now you have two separate disks: one with new employee files on it and one with old employee files on it.

REMOVE

REMOVE ***filespec*** **[*filespec* ...]** **REMOVE** ***devspec*** **[*devspec* ...]**	**Command**

Deletes *filespec* from the directory and frees the space allocated to it, or deletes *devspec* from the device table.

You can use REMOVE to delete a file that you don't need to use anymore. You can also REMOVE a device that is no longer needed.

Examples

```
REMOVE ALPHA/DAT:0 BREAKER/DAT:0 (ENTER)
```

deletes ALPHA/DAT and BREAKER/DAT from the directory on Drive 0 and frees all space allocated to them.

```
REMOVE MIDWEST/DAT.SECRET (ENTER)
```

deletes MIDWEST/DAT. If the file is protected at a level of RENAME or higher, the owner password must be used to remove the file. If you supply the user password, the error message "Illegal access attempted to protected file" is displayed. If you supply the wrong password, the error message "File access denied" is displayed.

```
REMOVE *LU (ENTER)
```

removes the user-created device *LU from the device table.

NOTE: A device can be removed only if it is pointed NIL in the device table. If a device is not pointed to NIL, it must first be reset with the RESET library command before it can be removed.

TRSDOS does not permit the removing of the system devices: *JL, *KI, *DO, *SI, *SO, and *PR. Attempting to remove these devices produces the error message "Protected system device."

Sample Use

Suppose you have a file of temporary employees that you hire for inventory. All of the temporary employees' files are in a file named EMPLOYEE/TEM. When you complete the inventory, you can remove this file with the command:

```
REMOVE EMPLOYEE/TEM (ENTER)
```

RENAME

Command

RENAME *filespec1* [TO] *filespec2*
RENAME *devspec1* [TO] *devspec2*

Changes a file's name and/or extension from *filespec1* to *filespec2*. It also changes a device name from *devspec1* to *devspec2*.

You can use RENAME to change the name of a file or a device.

If you wish to rename a file that contains an extension to a filename that does not contain an extension, you must include the slash (/) in the new file name. If you omit the slash, TRSDOS assumes the extension of the source filename for the destination filename.

RENAME does not change the file's password, contents, or position on the disk. (See the ATTRIB command to change the password.) If *filespec1* is password protected, the password must be specified or an error message will result.

RENAME does not change a device's routing, filtering, linking, or setting. *devspec1* must be an existing device, and *devspec2* must be an unused device name.

You cannot RENAME the system devices: *KI, *DO, *PR, *SI, *SO, and *JL.

Examples

```
RENAME TEST/DAT:0 TO OLD/DAT (ENTER)
```

renames TEST/DAT on Drive 0 to OLD/DAT.

```
RENAME TEST/DAT:0 TO REAL (ENTER)
```

renames TEST/DAT on Drive 0 to REAL/DAT. Since you did not specify an extension for *filespec2*, it defaulted to the extension on *filespec1* (DAT).

```
RENAME TEST/DAT:0 TO REAL/ (ENTER)
```

renames TEST/DAT on Drive 0 to REAL (without an extension).

```
RENAME DATA/NEW.SECRET:1 TO /OLD (ENTER)
```

renames the password protected DATA/NEW.SECRET on Drive 1 to DATA/OLD.SECRET. Since you did not specify a filename for *filespec2*, it defaulted to that of *filespec1*. RENAME does not change or delete passwords, so the password defaulted also.

```
RENAME *UD TO *TX (ENTER)
```

renames the device *UD to *TX.

Error Conditions:

If you attempt to rename a file to a filename that already exists on that diskette, TRSDOS displays a "Duplicate file name" error message.

If you specify an illegal filespec or attempt to use *devspec* and *filespec* in the same RENAME command, TRSDOS displays a "Specification error" message.

If you omit the destination *filespec2* or *devspec2*, TRSDOS issues a "Rename it to what?" error message. You must specify source and destination *filespecs* and *devspecs*.

REPAIR (REPAIR/CMD)

REPAIR :*drive*	**Utility**

Updates and modifies information on floppy disks produced by Radio Shack Model I operating systems and computers so TRSDOS Version 6 can use them.

drive is any floppy drive currently enabled in the system (except Drive 0).

After REPAIR is complete, you should be able to copy any file off the modified diskette.

Once a disk is modified by TRSDOS, the operating system that created it may not be able to read it.

TRSDOS 1.2 and 1.3 disks should NEVER be repaired. Use the CONV utility to copy programs from them.

Using REPAIR, you can convert the following list of operating system diskettes to TRSDOS Version 6 diskettes:

- Model I TRSDOS 2.0, 2.1, 2.2, 2.3, 2.3a

Examples

```
REPAIR :1 (ENTER)
```

updates information on the diskette in Drive 1 so that TRSDOS can use it.

Error Conditions

TRSDOS assumes that Drive 0 always contains a valid system disk. If you attempt to REPAIR a diskette in Drive 0, TRSDOS displays a "Can't REPAIR Drive 0" error message.

RESET

RESET *devspec* **RESET *filespec***	**Advanced Programmer's Command**

RESETs a system device to its normal condition in the DEVICE table. You can also use RESET to close a file that has not been properly closed.

Resetting a Device

A RESET *devspec* command removes any filtering, linking, or routing that has been set to the device. Any open disk file that is connected to the device is closed.

If *devspec* is a device you created (see the LINK and ROUTE library commands), it is pointed at NIL when reset.

If *devspec* is a system device (*KI, *DO, *PR, *SI, *SO, and *JL), it returns to its start-up condition when reset.

To see that *devspec* has been pointed at NIL or returned to its start-up condition, issue a DEVICE library command with the B parameter, and examine the device table that is displayed.

You can use the REMOVE library command to remove the device from the device table once it points at NIL.

Example

Suppose you have used the FORMS command to specify printer parameters, or you have filtered, linked, or routed *PR.

```
RESET *PR (ENTER)
```

returns *PR to its start-up condition and disconnects the printer filter.

Resetting a Filespec

You can RESET an open *filespec* that has not been properly closed.

Improperly closed files result when (1) your system loses power and files are left open, (2) you remove a disk from a drive and files are left open, or (3) you reset your system while files are open, or (4) a command aborts while files are open.

To see if any files on a disk are not properly closed, issue a DIR library command. Any file that appears with a question mark (?) after it needs to be RESET before you can access it. Receiving the error "File already open" may also indicate a file is not properly closed.

Example

Suppose that your system lost power and there is a file named PRINTER/DAT that is not properly closed.

```
RESET PRINTER/DAT (ENTER)
```

closes the file named PRINTER/DAT and lets you access it.

> **Advanced Programmer's Command**
> **ROUTE *devspec1*** [TO] ***devspec2***
> **ROUTE *devspec1*** [TO] ***filespec*** [(**REWIND**)]
> **ROUTE *devspec1* (NIL)**

Routes devspec1 to one of the following:

- another device (*devspec2*)
- a disk file (*filespec*)
- nothing (NIL)

You can use ROUTE to create a device. You can also use ROUTE to alter the flow of data from one device to another.

If *devspec1* does not already exist, ROUTE creates it.

To see how your devices are routed, use the DEVICE command. To return the non-system devices to their normal start-up state, use the RESET and REMOVE commands.

Examples

```
ROUTE *PR *DO
```

routes the printer (*PR) to the video display (*DO). All data normally sent to the printer will be displayed on the screen.

```
ROUTE *PR TO PRINTER/DAT
```

routes the printer (*PR) to a disk file (PRINTER/DAT). All data normally sent to the printer will be stored in a disk file named PRINTER/DAT. If PRINTER/DAT already exists, the data is appended to the end of the file.

```
RESET *PR
```

closes the PRINTER/DAT file and any subsequent output to *PR goes to the printer. The PRINTER/DAT remains open until you execute the above command.

```
ROUTE *PR TO PRINTER/DAT (REWIND)
```

routes the printer to PRINTER/DAT. If PRINTER/DAT already exists, the system "rewinds" the file to the beginning. The contents of the file will be replaced with the new printer data.

```
ROUTE *PR (NIL)
```

routes *PR to NIL. TRSDOS ignores all output to the printer.

```
ROUTE *DU TO TEST/TXT:1
```

routes a user device (*DU) to a disk file named TEST/TXT in Drive 1.

If you ROUTE *CL to a file and you are receiving data from a communications line (*CL), you might lose data if it is coming in at a high speed. If so, use CREATE to preallocate file space before using ROUTE. This makes data loss less likely because the system no longer has to spend the time allocating more space.

Error Conditions

If you misspell *devspec2* or specify a device that doesn't exist TRSDOS issues the error message "Device not available." Check your spelling and try again.

If there are too many devices or routes, TRSDOS issues a "No memory space available" error message. TRSDOS also issues this error message if an application program sets Bit 0 of CFLAG$. The application program must reset the bit in CFLAG$. See the *Model 4/4P Technical Reference Manual* for additional information on CFLAG$.

Sample Use

Suppose you want to route a report-producing program to a file (instead of printing the report). Issue the command:

```
ROUTE *PR TO REPORT/DAT (ENTER)
```

Now you can run the program and the report that it produces is routed to the file REPORT/DAT. This means that you can print the report in the file REPORT/DAT whenever you want by using the LIST command.

NOTE: Remember, you must reset *PR before listing the file so that it will be properly closed.

RUN

[RUN [(X)]] ***filespec*** [***command text***]	**Command**

Loads a program named *filespec* into memory and executes it.

Typing RUN is optional. You can load and execute a program from the TRSDOS Ready prompt by simply typing in the name of the program (without the RUN).

The default extension for *filespec* is CMD.

X executes a program from a non-system disk for the single drive user.

Command text is an optional value which the program you specified may require.

When running a program, observe the following address restrictions:

RUN   *filespec* must load above X'25FF'.
RUN (X)   *filespec* must load above X'2FFF'.

Examples

```
RUN CONTROL/CMD (ENTER)
```

loads a program named CONTROL/CMD and executes it.

```
CONTROL/CMD (ENTER)
```

loads a program named CONTROL/CMD and executes it.

```
RUN PROG (ENTER)
```

loads a program named PROG/CMD and executes it. Since you did not supply a file extension, it defaulted to /CMD.

```
RUN (X) TRADERS/CMD:0 (ENTER)
```

loads TRADERS/CMD from a non-system disk. First you are prompted with the message:

```
Insert SOURCE disk <ENTER>
```

Insert the disk containing the program into Drive 0 and press (ENTER). After the program is loaded into memory, you are prompted with.

```
Insert SYSTEM disk <ENTER>
```

Insert the system disk back into Drive 0 and press (ENTER); program execution begins.

SET

Advanced Programmer's Command

SET *devspec* [TO] *filespec* [USING] [(*parameters*)]

Loads a driver of filter program into memory and sets it to a device.

devspec can be a system device or a *phantom device* (non-existing device). If you specify a phantom device, you must use the FILTER command to connect the phantom device to a system device.

filespec can be a TRSDOS filter program, your own filter program, or a TRSDOS driver program.

parameters are values sent to the driver of filter program. They are totally independent of the SET command and determined only by the needs of your driver of filter program.

A driver program channels data to or from a device. If it is outputting to a device, it converts data to the device's format. If it is inputting from a device, it converts the data to the computer's format.

A filter program filters data before it is sent out or after it is received.

Once the device is SET, it remains SET until it is RESET. You cannot SET an active device.

See Appendix I for a complete list of TRSDOS filters and drivers. See Appendix K for more examples of setting devices, drivers and filters.

Example

Within TRSDOS is a driver program that sends printer output to the parallel port. Suppose you write a driver program named SERIAL/DVR that sends printer output to the serial port.

```
SET *SP TO SERIAL/DVR (ENTER)
```

loads SERIAL/DVR into memory and sets it to the device *SP.

```
ROUTE *PR TO *SP (ENTER)
```

routes data going to the printer to the device *SP. Now any input to the printer goes to the SERIAL/DVR program. The SERIAL/DVR program, in turn, sends the output to the serial port.

By using a LINK command instead of the ROUTE command, data sent to *PR is sent to the TRSDOS parallel printer driver. It is also sent (via the LINK) to the serial driver and then to the serial printer.

Example

Suppose you write a filter program named TRAP/FLT to change some characters sent to *DO. the video display.

First, you need to load your TRAP/FLT program and set it to a phantom device (this example uses *LC as the name of the phantom device):

```
SET *LC TO TRAP/FLT
```

This causes *LC to point to TRAP/FLT.

Then, you need to use *LC (which points to the TRAP/FLT program) to filter the data output to the video display:

```
FILTER *DO *LC (ENTER)
```

Now, all data output to the video display is filtered through your filter program.

```
SET *DU KSM/FLT USING FILEDAT/KSM (ENTER)
FILTER *KI *DU (ENTER)
```

loads the Keystroke Multiply filter into memory and sets it to the keyboard.

Error Conditions

If there are too many devices or routes, TRSDOS issues a "No memory space available" error message. TRSDOS also issues this error message if an application program sets Bit 0 of CFLAG$. The application program must reset the bit in CFLAG$. See the *Model 4/4P Technical Reference Manual* for additional information on CFLAG$.

SETCOM

SETCOM [(*parameters*)]	**Advanced Programmer's Command**

Adjusts the parameter values of the RS-232-C driver program COM/DVR. Before you can use SETCOM, you have to install the driver using the SET command (see Appendix I).

You can use SETCOM to adjust your computer so that it can communicate with another computer or a piece of computer equipment.

The RS-232-C port lets you communicate with:

- another computer
- a modem
- a serial printer

You can include *parameters* to configure the RS-232-C port and establish line conditions. If you omit parameters, TRSDOS displays three lines of information about the current parameter settings, and input and output line conditions.

The *parameters* are:

DEFAULT returns all parameters to their start-up values.
BAUD=*number* sets the BAUD rate to any supportable rate. *number* can be 110, 135, 150, 300, 600, 1200, 2400, 4800, 9600. The default value for BAUD is 300.
WORD=*number* sets the word length to *number*. *number* can be 5, 6, 7, or 8. The default value for *number* is 7.
STOP=*number* sets the *number* of stop bits per word. *number* is either 1 or 2. The default value for *number* is 1.
QUERY prompts you for each parameter.
BREAK=*value* sets the character that COM/DVR recognizes as a BREAK function. *value* may also be Y or N. Y sets the BREAK value to Hexadecimal 03 (CTRL C). N sets the BREAK value to Hexadecimal 80 (BREAK). If you specify 0 for value, COM/DVR does not recognize any character as a BREAK.
PARITY=*switch* sets parity. *switch* can be ON, OFF, EVEN, or ODD. You must enclose EVEN and ODD in quotes.
ON enables parity and retains its previous value, EVEN or ODD.
OFF disables parity.
EVEN enables parity and establishes EVEN parity
ODD enables parity and establishes ODD parity.

BREAK cannot be abbreviated.

Examples

```
SETCOM (ENTER)
```

displays the current configuration of the RS-232-C port in the following format:

```
RS232 parameters: Baud=300, Word=7, Stop=1, Parity=EVEN, Break=X'03'
Output control: DTR=ON, RTS=OFF
Input control: RI=IGNORE, DSR=IGNORE, CD=IGNORE, CTS=IGNORE
```

```
SETCOM (BAUD=300,WORD=8,STOP=1, PARITY=NO) (ENTER)
```

configures the RS-232-C using the values specified. Notice that PARITY is specified as NO.

Technical Information

This command allows you to set the parameters to values that match any other RS-232-C devices. The receiving side of the driver is interrupt driven and contains an internal one-character buffer to prevent loss of characters during disk I/O and other lengthy operations. The system usually uses the *CL devspec to communicate with the RS-232-C port.

TRSDOS can lose RS-232 characters during disk access if SMOOTH is on. See the SYSTEM command.

If you are using a serial printer, (1) use SET to set *CL to COM/DVR (see Appendix I), (2) use SETCOM to set the proper parameters, and (3) use the command:

```
ROUTE *PR TO *CL (ENTER)
```

to direct output to the RS-232-C (rather than the standard parallel port). Tandy printers do not require this procedure since they use the parallel printer port.

The line condition parameters let you set up the conventions required by most communicating devices.

The RS-232-C line output *parameters* are:

DTR=*switch* Data Terminal Ready
RTS=*switch* Request To Send

The RS-232-C line input *parameters* are:

DSR=*switch* Data Set Ready
CD=*switch* Carrier Detect
CTS=*switch* Clear To Send
RI=*switch* Ring Indicator

switch is either YES or NO. You cannot abbreviate any of the RS-232-C line parameters.

As specified by standard RS-232-C conventions, a TRUE condition means a logical 0, or positive voltage. A FALSE condition means a logical 1, or negative voltage.

DTR and RTS can be set to a constant TRUE by specifying the YES switch. If DSR, CD, CTS, or RI is specified YES, the driver observes that signal and waits for a TRUE condition before sending each character. If specified NO, the driver waits for a false condition before sending a character. If not specified, that signal is ignored.

The BREAK parameter allows you to set a logical BREAK character.

This is useful in "host" type applications. The BREAK parameter causes the serial driver to set the system break bit whenever a modem break (extended null) or an ASCII logical BREAK is received. The system pause bit is set whenever the hex code 60 is received. The system enter bit is set whenever a carriage return (0D) is received.

The default for BREAK is 3, so a (CTRL)(C) sets the break bit. Use BREAK=*value* to set another character as the logical break.

Technical Examples

```
SETCOM (BREAK) (ENTER)
```

configures the RS-232-C port to the default values. Specifying BREAK with no value assigns the default value of 3 as the logical break value.

```
SETCOM (CTS) (ENTER)
```

configures the RS-232-C port to the default values. Because CTS is specified, the driver looks at the CTS line for a TRUE condition before it sends a character.

Error Conditions

If you do not install COM/DVR in memory with the SET command before you issue a SETCOM command, TRSDOS displays a "COM/DVR not installed" error message.

Sample Use

Suppose you want to log-on to CompuServe. First, you have to SET *CL to COM/DVR, and then you have to use SETCOM to set the parameters of the RS-232-C port so that your computer can communicate with CompuServe. See the Logging-On to CompuServe section in the COMM utility description.

SETKI

SETKI [(*parameters*)]	**Advanced Programmer's Command**

Sets keyboard repeat *parameters*. If you do not specify a *parameter*, the current delay and repeat rate settings are displayed.

You can use SETKI to adjust how your keyboard reacts when you press a key.

The parameters are:

DEFAULT returns the parameters to their start-up values.
RATE=*number* sets the repeat rate as *number*. *number* is any number greater than or equal to 1. *number* equals 2 when the system is started or reset.
WAIT=*number* sets the initial delay between the time a key is first pressed and the first repeat of that key as *number*. *number* is any number greater than or equal to 10. *number* equals 22 when the system is started or reset.
QUERY prompts you to enter new values for RATE=*number* and WAIT=*number*.

Examples

```
SETKI (WAIT=15) (ENTER)
```

sets the delay rate to 15.

```
SETKI (ENTER)
```

displays the current delay and repeat rate settings in the format:

Wait=15, Rate=2

Note: Both the RATE and WAIT parameters use modulo 128. For example, entering 138 has the same effect as entering 10.

SPOOL

SPOOL [*devspec*] [TO] [*filespec*] (*parameters*)	**Command**

establishes a First-In, First-Out buffer for a specified device (usually a line printer).

You can use SPOOL to print data while you perform other operations on your computer (such as running a BASIC program).

If you do not specify *devspec*, it defaults to *PR.

The *parameters* are:

NO turns off the spooler and resets *devspec*.
MEM=*number* specifies *number* as the amount of memory buffer (in K) to be used by the spooler. The value of *number* is 1 - 32.
BANK=*number* selects one of three 32K banks of memory to be used as the spool buffer. *number* can be a 0, 1, or 2. The default value of *number* is 0.
DISK=*number* specifies *number* as the amount of disk space (in K) to be used by the spooler. The value of *number* cannot be larger than the amount of available space (in K) on the disk. For each K of disk space specified, 16 bytes of RAM are automatically reserved for the spooler's use.
PAUSE temporarily suspends output to *devspec*.
RESUME restarts *devspec* after a PAUSE.
CLEAR clears the spool buffer.

How Data Is Spooled To a Device

All data sent to *devspec*, such as a printer, is placed in an output buffer where it waits until the device is again available to accept the data.

There are two kinds of output buffers: memory and disk. You can set up a spooler that uses memory or disk or both.

The minimum space allocation for the memory buffer depends on which BANK you select. If you specify BANK 0, a minimum of 1K (1024 bytes) is allocated for the memory buffer. If you specify BANK = 1 or BANK = 2, the entire 32K bank is automatically used for the memory buffer.

If you specify both buffers, data is sent first to the memory buffer. When the memory buffer is full, the data is sent to a disk buffer named *filespec*, where it waits to be sent to the device. If you specify a memory buffer only, data is sent to a memory buffer until the device is ready to accept it.

When you specify *filespec*, you may also use the DISK parameter to specify the amount of disk space to be used by the spooler. TRSDOS creates a file of the size specified. If you do not specify DISK=, approximately 5K of disk space is automatically allocated to *filespec*.

To prevent TRSDOS from allocating any disk space to SPOOL, specify DISK=0.

filespec remains open as long as SPOOL is on. Do not REMOVE this file or remove the disk from the drive without closing the file (by issuing a SPOOL *devspec* (NO) command).

You cannot issue a SYSGEN library command if the spooler is on.

Once the spooler is turned off, you can turn it on again. The same memory locations are used, but the following restrictions apply:

- The original parameters are not affected by turning *devspec* off and then on.
- Any parameter specified the second time cannot exceed the memory or disk parameters originally given. If it does, an error occurs.

Examples

```
SPOOL *PR TO TEXTFILE:0 (MEM=5,DISK=15) (ENTER)
```

allocates 5K of memory and 15K of disk space in a file named TEXTFILE/SPL on Drive 0. Since you did not specify an extension to TEXTFILE, it defaulted to /SPL.

Any output for the printer is buffered and sent to the line printer (*PR) as fast as the printer can accept the characters. If the 5K memory buffer is filled, the data is written to the disk file TEXTFILE/SPL on Drive 0.

```
SPOOL *PR (BANK=1,DISK=0) (ENTER)
```

creates a 32K memory buffer for data sent to *PR. Any output for the printer is sent to the memory buffer and then spooled to *PR when it is available to accept the data. Since the parameter DISK= is specified without any size, none of the spooled data is sent to a disk file.

If the memory buffer is filled, TRSDOS does not process any more printer data until *PR has printed enough data to bring the number of characters waiting to be printed below 32K (the size of the memory buffer).

```
SPOOL (CLEAR) (ENTER)
```

clears the information in the spool buffer.

```
SPOOL *PR (NO) (ENTER)
```

turns off the spooler and closes the associated disk file. Any filtering, linking, setting, or routing done to *PR is reset.

You cannot close the disk file by issuing a RESET or REMOVE library command. SPOOL must be turned off to close the file.

Sample Use

Since most programs produce reports faster than the printer can print the data, you can use SPOOL to let the programs run at top speed without having to stop and wait on the printer. That is, while the first program's report is still printing, you can begin executing a second program.

Error Conditions

Some application programs do not honor HIGH$ and write over a portion of the SPOOL program. If this happens, the error message "Can't locate SPOOL in memory" appears. In the future do not use that application program with SPOOL.

If you issue a SPOOL (OFF) command when SPOOL is already off, the error message "Spool is not active" appears. If you attempt to change the parameters of SPOOL after SPOOL is active, the error message "Spool is already active" appears. You must turn SPOOL off and turn it back on with the new parameters.

If you load another module into memory after you turn the spooler off, and you attempt to reload the spooler with different parameters, TRSDOS issues a "Cannot reinstall with altered parameters" error message. You can re-activate the spooler with the original parameters. If you want to alter the spool parameters, you must reset the system. If you still need the other module, reload it and reload the spooler with different parameters.

If you load another program that uses high memory after you load SPOOL, TRSDOS may not be able to release memory it is using. The error message "Can't reclaim memory space" appears. You must reset the system if you need the space.

If you attempt to use Banks 1 or 2 on a 64K system or if you request a bank that is already in use, the message "Requested bank in use" appears.

If there is not enough memory available to set up the routes that the spooler uses, TRSDOS displays a "No memory space available" error message. Remove any unused logical devices.

TRSDOS also issues this error message if an application program sets Bit 0 of CFLAG$. The application program must reset the bit in CFLAG$. See the *Model 4/4P Technical Reference Manual* for additional information on CFLAG$.

The driver that is connected to the devspec specified in a SPOOL command must be an output device driver. If you specify an input device the error message "Device driver incompatible" appears.

Advanced Programmer's Command
SYSGEN [([*switch*] [,] [**DRIVE**=*drive*])]

Creates a configuration file on *drive* to store information about your system.

You can use SYSGEN to create a file of current device and driver configurations that you want TRSDOS to execute each time you restart the system.

If you do not specify *drive*, it defaults to Drive 0.

The *switch* is either YES or NO.

If you specify *switch* as YES, then your system creates a configuration file. If you don't specify *switch*, then YES is assumed.

If you specify *switch* as NO, then your system removes the configuration file. However, your system's current configuration does not change until you reset your computer.

When you issue a SYSGEN command, all current device and driver configurations are stored in a file named CONFIG/SYS. The file is invisible in the directory. You can see it by using the INV parameter in the CAT and DIR commands.

Each time you reset your computer, TRSDOS loads the CONFIG/SYS file into memory. While this program is loading, TRSDOS displays the message ** SYSGEN ** in the lower left corner of the display.

If you do not want TRSDOS to load CONFIG/SYS, hold down the (CLEAR) key when you reset the system. See the BOOT command for additional information on booting your system.

Note: CONFIG/SYS files that were created using previous versions of TRSDOS, Version 6.0 or 6.1, cannot be used with TRSDOS Version 6.2. You cannot copy a configuration file to a disk. You must use SYSGEN to create a new CONFIG/SYS file for Version 6.2.

When you start up or reset your computer, it is configured before any AUTO command executes.

The configuration file CONFIG/SYS contains:

- All active background tasks (such as CLOCK, DEBUG, TRACE, etc.).
- All filtering, linking, routing, and device setting (including RS-232-C and KI settings).

- All programs that were loaded into high memory above HIGH$. All memory from HIGH$ to the top of memory is written to CONFIG/SYS. HIGH$ can be set with the MEMORY command or with the @HIGH$ supervisor call. (See the *Model 4/4P Technical Reference Manual*.)
- The present state of the VERIFY library command (YES or NO).
- All Device Control Blocks. (See the *Model 4/4P Technical Reference Manual* for more information.)
- The present state of the CAPS lock for the keyboard.

Examples

```
SYSGEN (YES) (ENTER)
```

creates a configuration file on Drive 0 and writes the system configuration to it.

```
SYSGEN (NO) (ENTER)
```

removes the configuration file from Drive 0.

Error Conditions

If you use a SYSGEN command in a JCL file, TRSDOS issues a "Command invalid during <DO> processing" error message.

If you specify a value for DRIVE that does not contain a system diskette, TRSDOS displays a "Warning: Target drive contains no system" error message.

If you attempt to sysgen while a device is routed to a file, TRSDOS displays a "Can't while route-to-file is active" error message. You must remove all routes before you sysgen.

Sample Use

Suppose you want to create a file of commands that automatically execute each time you startup TRSDOS.

Issue the commands:

```
TIME (CLOCK=YES) (ENTER)
SYSTEM (TRACE=YES) (ENTER)
SYSGEN (YES) (ENTER)
```

to create a CONFIG/SYS file that contains CLOCK and TRACE information.

Advanced Programmer's Command
SYSTEM (*subcommand[parameters*])

Allows you to change the configuration of your TRSDOS system.

You can use the SYSTEM command to customize portions of TRSDOS to function differently when you boot or reset the system. When you make these changes, you can store them on the diskette in Drive 0 with the SYSGEN command. Every time you boot or reset TRSDOS with that diskette, your changes are in effect rather than the original TRSDOS values. For example, you can use the BLINK subcommand to change the cursor from a blinking cursor to a non-blinking cursor or to another character.

> **Note:** Certain SYSTEM subcommands do not require you to store the changes on diskette. The changes are stored automatically as soon as you enter the command. The subcommands included are: BSTEP, CYL, DATE, HERTZ5, HERTZ6, RESTORE, and TIME.

You can use the SYSTEM command with subcommands to set or change the disk drive configuration, load driver routines into high memory, and turn on or off keyboard, video, and hardware functions. Some SYSTEM subcommands use portions of high memory. Each of the subcommands and their parameters are described in more detail below.

The DEVICE or MEMORY commands display the current configuration of your TRSDOS system.

If an application program sets Bit 0 of CFLAG$, TRSDOS issues a "No memory space available" error message when you attempt to use the SYSTEM subcommands. The application program must reset the bit in CFLAG$. See the *Model 4/4P Technical Reference Manual* for additional information on CFLAG$.

Subcommands

ALIVE displays a changing character in the screen's upper right corner.

    SYSTEM (ALIVE[=*switch*])

switch can be YES or NO. If you omit *switch*, TRSDOS assumes YES and displays the changing character. NO disables the ALIVE subcommand.

The changing character indicates that the task processor is running. The character may continue to move even when the SYSTEM (TRACE) command stops.

During FORMAT operations and disk I/O, the character stops moving. Otherwise, if the character is not moving, the system is "hung-up."

BLINK changes the cursor character.

SYSTEM (BLINK*parameter*)

Parameter may be:

=YES returns the cursor to its default character, a blinking ASCII 95 (X'5F').
=NO changes the blinking cursor to a non-blinking cursor.
=*number* changes the cursor to the ASCII character you specify with number.
,LARGE changes the cursor to the character ASCII 191.
,SMALL changes the cursor to the character ASCII 160.

If you omit *parameter*, BLINK changes the cursor to its default character, a blinking ASCII 95 (X'5F').

Keep in mind that characters above ASCII 127 change if reverse video is enabled. Consider your choice of characters carefully.

BREAK enables or disables (BREAK).

SYSTEM (BREAK[=*switch*])

switch may be YES or NO. If you omit *switch*, TRSDOS assumes YES and enables (BREAK). YES enables (BREAK) if it was disabled by the AUTO**command* library command.

Pressing (BREAK) has no effect after you execute a (BREAK=NO) command.

BSTEP establishes the default bootstrap step rate TRSDOS uses when formatting.

SYSTEM (BSTEP=*number*)

number can be a number in the range 0-3 indicating the following step rate:

0	6 milliseconds
1	12 milliseconds
2	20 milliseconds
3	30 milliseconds

The Drive 0 diskette must be write-enabled when changing the bootstrap step rate. TRSDOS stores the value on logical Drive 0. If you change the diskette in Drive 0 or change the logical Drive 0 with the SYSTEM (SYSTEM) command, TRSDOS assumes the bootstrap rate on the new system disk.

DATE enables or disables the date prompt when you turn on your computer.

SYSTEM (DATE[=*switch*])

switch may be YES or NO. If you omit *switch*, TRSDOS assumes YES and enables the date prompt. NO disables the date prompt when you turn on your computer.

We recommend that you do not disable the date prompt. TRSDOS uses the date when accessing and creating files, making backups, and formatting disks.

DRIVE allows you to configure your system's logical drives.

SYSTEM (*parameter*[,*parameter*])

The parameters are:

DRIVE =*drive* specifies any valid drive number that you are changing the default values for. If you include DRIVE, use it only one time as the first parameter in the command line.

You must specify the DRIVE with DRIVER, ENABLE, and DISABLE.

DRIVER="*device driver*" specifies the name of the disk driver for the drive number you specify with the DRIVE parameter.

DISABLE removes access to the drive you specify with the DRIVE parameter. If you try to access a drive after you have disabled it, TRSDOS displays an "Invalid drive number" error message.

ENABLE allows access to a disabled drive.

If you omit DRIVE with CYL, DELAY, STEP, and WP, TRSDOS sets these values on all enabled drives.

CYL=*number* specifies the default number of cylinders for the FORMAT utility. *number* can be a value in the range 35 to 96. TRSDOS writes CYL to the diskette in Drive 0. If you change the diskette in Drive 0, TRSDOS takes the value for CYL from the current Drive 0 diskette.

DELAY=*switch* sets the delay time for diskettes. Delay time is the maximum length of time TRSDOS allows between drive motor-on and the first attempted access (read or write) of that drive. *switch* may be YES or NO. If you specify NO, TRSDOS allows the standard delay time, .5 seconds. If you specify YES, TRSDOS sets delay time to 1 second.

STEP=*number* sets the step rate. Step rate is the length of time for the read-write head to move from one cylinder to another. The step rate remains in effect until the system is re-booted, turned off or sysgened. *number* may be a number in the range 0 to 3 and represents the following rates in milliseconds:

Number	Milliseconds
0	6
1	12
2	20
3	30

WP=*switch* sets the Write Protect status. *switch* may be YES or NO. If you omit *switch*, TRSDOS assumes YES. If you specify YES, you cannot write to the disk, although you can still read from it. If you specify NO, you can write to and read from the disk (assuming the disk is not hardware write protected).

FAST sets the system speed at 4 Megahertz (MHz), your computer's speed.

SYSTEM (FAST)

Timing loops in TRSDOS Version 1 programs may require you to use the SLOW subcommand to run the system at the Model III speed of 2 MHz. These programs may not function properly at 4 MHz. Use the FAST subcommand to reset the system to 4 MHz after executing a SLOW subcommand.

GRAPHIC informs TRSDOS that the printer you are using has the capability of reproducing TRS-80 graphics characters during screen print.

SYSTEM (GRAPHIC[=*switch*])

switch may be YES or NO. If you omit *switch*, TRSDOS assumes YES, indicating that your printer has the capability to reproduce graphics characters.

When you press CTRL and :, TRSDOS sends a duplicate copy of the screen to the printer. Normally, TRSDOS prints all characters that have ASCII values greater than X'7F' as periods (.). When you execute a GRAPHICS=YES statement, TRSDOS prints these characters as their actual graphic characters. You must have a graphics printer to print these characters.

HERTZ patches your system to run at 50 or 60 Hertz (Hz).

SYSTEM (HERTZ*number*)

number can be 5 to specify 50 Hertz, or 6 to specify 60 Hertz. TRSDOS stores this value on the logical Drive 0. There must not be a space between HERTZ and *number*.

Before you use the HERTZ subcommand, see Appendix L for additional information.

RESTORE enables or disables the restoring of all drives to Track 0 at startup.

SYSTEM (RESTORE[=*switch*])

switch may be YES or NO. If you omit *switch*, TRSDOS assumes YES and restores all drives.

TRSDOS assumes that the drives are at Cylinder 0 when you startup or reset the system. The RESTORE subcommand causes TRSDOS to force the hardware to Cylinder 0 as well. This decreases initial disk access time.

SLOW sets the system speed at 2 Megahertz (MHz), Model III speed.

SYSTEM (SLOW)

Timing loops in TRSDOS Version 1 programs may require you to use the SLOW subcommand to run the system at the Model III speed of 2 MHz. These programs may not function properly at your computer's speed of 4 MHz. Use the FAST subcommand to reset the system to 4 MHz after executing a SLOW subcommand.

SMOOTH allows smoother disk access by disabling interrupts earlier.

SYSTEM (SMOOTH[=*switch*])

switch may be YES or NO. If you omit *switch*, TRSDOS assumes YES and activates SMOOTH.

When SMOOTH is active, TRSDOS disables interrupts when reading data from a floppy disk. This increases disk access speed. However, the type-ahead function depends on interrupts occurring at regular intervals. Disabling the interrupts can cause a loss of keystrokes and RS-232 characters during disk I/O. When Smooth is active, the time-of-day clock is not accurate.

SYSRES loads TRSDOS system overlays into high memory.

SYSTEM (SYSRES=*number*)

number specifies the overlay TRSDOS loads into memory. *number* may be 1, 2, 3, 4, 5, 9, 10, 11, or 12. You can only specify one system overlay per command line.

Every time TRSDOS needs to access a system overlay, TRSDOS must load that overlay into memory. Loading the overlays into memory increases processing time because they are immediately available.

If you load the overlays into high memory, you can remove them from your system diskette with the PURGE command. This leaves more room available on your diskette for data and program files.

SYS0, SYS1, SYS2, and SYS3 must be on the booting disk if you load a configuration file with the SYSGEN command. See the *Model 4/4P Technical Reference Manual* for additional information.

Use the DEVICE command to display which overlays are currently in memory.

SYSTEM assigns a drive other than Drive 0 as your system drive.

SYSTEM (SYSTEM=*drive*)

drive may be any valid drive in your system. The drive that you specify becomes logical Drive 0. The original Drive 0 then becomes the logical drive number specified.

There must be a system disk in *drive* when you execute a SYSTEM subcommand. Every time you execute a SYSTEM subcommand, the

logical drive numbers of Drive 0 and *drive* change. You can repeat this command as many times as you want. Be sure to remember which drive is assigned to which logical drive number.

Note: A system disk should contain these files:

SYS0/SYS
SYS1/SYS
SYS2/SYS
SYS3/SYS
SYS4/SYS
SYS10/SYS
SYS12/SYS

and optionally the command libraries:

SYS6/SYS
SYS7/SYS
SYS8/SYS

TIME enables or disables the time prompt when you start up your system.

SYSTEM (TIME[=*switch*])

switch may be YES or NO. If you omit *switch*, TRSDOS assumes YES and enables the time prompt. NO disables the prompt after you have executed a TIME=YES command.

If TRSDOS does not prompt you for the time when you startup your system, the system clocks starts at 00:00:00 each time you start up the system.

TRACE displays the contents of the program counter (PC) in the upper right corner of the video display.

SYSTEM (TRACE[=*switch*])

switch may be YES or NO. If you omit *switch*, TRSDOS assumes YES and displays the program counter. NO turns off the display after a TRACE=YES command.

The TRACE subcommand is useful when debugging assembly-language programs. The value that TRACE displays is the hexadecimal value of the Z-80 program counter and is constantly updated by a low priority background task. The program counter contains the address of the next instruction to be executed. See the @ADTSK SVC in the *Model 4/4P Technical Reference Manual* for additional information.

TYPE enables or disables the keyboard type-ahead feature.

SYSTEM (TYPE[=*switch*])

switch may be YES or NO. If you omit *switch*, TRSDOS assumes NO and disables the type-ahead feature. Use TYPE=YES to re-enable the feature.

TAPE100

TAPE100 [***file***] [(***parameters***)] **TAPE100** [***file1*** [TO] ***file2***] [(***parameters***)]	**Utility**

Lets TRSDOS (1) read a cassette tape file and write it to a disk file, or (2) read a disk file and write it to a cassette tape.

You can use TAPE100 to read files from cassette tape as well as from your computer's disks.

The cassette tape must have been made with the Model 100 computer.

file, file1, and *file2* are each either a TRSDOS filespec or a Model 100 filename.

A Model 100 filename is 1 - 6 alphanumeric characters long and it must begin with a letter. For example, ACCT61, LETTER, and ABFILE can be Model 100 filenames.

If you do not specify *file* or *file1* and *file2*, you will be prompted to enter the source and destination filespecs if the operation is a WRITE, or just the destination filespec if the operation is a READ.

The *parameters* are:

READ specifies that you want to read a file (*file* or *file1*) from tape and write it to a file (*file* or *file2*) on disk. If you specify READ, you do not have to specify *file1*. TRSDOS simply reads the first text file it sees on the tape.

WRITE specifies that you want to read a disk file (*file* or *file1*) and write it to a tape as *file* or *file2*.

If you do not specify the READ or WRITE parameter, you will be prompted for it.

Examples

```
TAPE100 PRNTER TO PRINT/DAT:0 (READ) ENTER
```

TRSDOS reads the Model 100 file PRNTER and writes it to the disk in Drive 0 as PRINT/DAT.

```
TAPE100 ACCTING/TXT:1 (READ) ENTER
```

TRSDOS reads the first text file it finds on a Model 100 tape and writes it to the disk in Drive 1 as ACCTING/TXT.

```
TAPE100 WEST/DAT:0 TO WESTRN (WRITE) ENTER
```

TRSDOS reads the Drive 0 disk file WEST/DAT and writes it to a file on a Model 100 tape named WESTRN.

Error Conditions

Any file (disk or tape) must fit in available memory or the error message "File too large to fit in available memory" appears.

TIME

TIME [*hh:mm:ss*] [(*parameter*)]	**Command**

You can use TIME to see the current time. You can also use it to reset the time.

If you specify *hh:mm:ss*, TRSDOS resets the time. If you do not specify it, TRSDOS displays the current time.

The parameter is:

CLOCK[=YES¦NO] turns the clock display on or off. YES is the default.

hh must be a value in the range 00 to 24, and *mm* and ss must be a value in the range 00 to 59. If you specify *hh:mm:ss*, you must specify valid numbers for all three values. Specifying values outside these ranges or not in the proper format results in the error "Bad time format."

You may use any of the ASCII characters in the range 32 (X'20') through 39 (X'27'), 41 (X'29') through 47 (X'2F'), and 58 (X'3A'), to separate hours, minutes, and seconds. See Appendix C for a complete list of ASCII character codes.

The real time clock turns off while TRSDOS does some of its disk I/O functions, such as BACKUP and FORMAT, so do not depend on the clock for constantly accurate time and date information.

You can enable and disable the prompt for time on power-up or reset with the SYSTEM (TIME=) command.

Examples

`TIME` (ENTER)

displays the real time of the system. The clock is reset to 00:00:00 every time you power up.

`TIME (CLOCK=YES)` (ENTER)

displays the real time clock in the upper right corner of the screen. Note: CLOCK will print over whatever TRSDOS attempts to print at the location occupied by the clock display.

`TIME 12:29:34` (ENTER)

sets the clock to 12:29:34 p.m. The latest acceptable time is 23:59:59, as the clock runs in the 24-hour mode. When the clock reaches 23:59:59, the date is automatically updated.

The time lag between pressing (ENTER) and the time set on the clock is approximately 2 seconds. So, when setting the clock with the correct time, remember to adjust for the 2-second time lag.

TOF

TOF	Command

Sends a top-of-form character (X'0C') to the printer.

TOF causes the printer to advance to the top of the next page before printing. You can use TOF with any printer.

If your printer cannot interpret a top-of-form character, you must use the FORMS filter with FFHARD off (the default setting) to use this command. All Tandy DMP series printers can interpret top-of-form characters.

See FORMS for additional information on FFHARD.

Example

```
TOF (ENTER)
```

sends a top-of-form character to the printer causing the printer to advance to the top of the next page.

Command

VERIFY [(*switch*)]

Controls the verify function.

You can use VERIFY to assure you that data was properly written to a disk.

When VERIFY is on, TRSDOS reads the data it writes to the disk to verify that the data is readable.

The *switch* is either ON or OFF.

A TRSDOS floppy disk system starts up with VERIFY off. A hard disk system starts up with VERIFY on.

Although having the VERIFY switch turned on provides a reliability check during disk I/O, it also increases overall processing time when you write to a disk file. You must determine if the increase in reliability warrants the increase in processing time.

All disk writes are automatically verified during any BACKUP utility function, whether or not the VERIFY switch is on.

The state of the VERIFY command can be saved in the configuration file with the SYSGEN library command. (You can check the present status of VERIFY using the DEVICE command.)

Examples

```
VERIFY (ON) (ENTER)
```

turns on the verify function.

```
VERIFY (OFF) (ENTER)
```

turns off the verify function.

```
VERIFY (ENTER)
```

turns on the verify function.

Sample Use

Suppose you are writing a tax file named TAX/TXT to disk and it is extremely important that the information in the file be correct.

Using VERIFY causes TRSDOS to produce an informative message when data written to TAX/TXT is written incorrectly. An informative message could indicate that the disk needs to be replaced or the drives need to be cleaned.

NOTES:

NOTES:

Part II/ Model 4 BASIC

Introduction

This part of the manual is about the BASIC language. Model 4 BASIC is an "interpreter." When you run a program, it executes each statement one at a time. This makes it quick and easy to use. It also allows you to take advantage of many of TRSDOS Version 6's features, such as:

- Faster running programs
- Better graphics capabilities
- More print positions on the screen

About Part II

Part II of this manual, like Part I, is for reference. It is not a tutorial. We assume you already know BASIC.

> Section III — Operations. This section shows how to load BASIC. It also demonstrates how to write, run and save a BASIC program on disk.
>
> Section IV — The BASIC Language. This section includes a definition for each of BASIC's keywords (statements and functions) in alphabetical order. In addition, it shows how to write a program to store data on disk.

If you have read *Getting Started with TRS-80 BASIC* (sold separately), you need to know the differences between Model III BASIC and Model 4 BASIC. Appendix E shows these differences. These differences will often prevent a BASIC program written for TRSDOS Version 1 (the Model III's operating system) from running under TRSDOS Version 6 (your computer's operating system), unless the program is modified. You also need to know how to use "disk files." This is explained in Chapter 5.

NOTE: From now on, "BASIC" (used alone) refers to Model 4 BASIC.

Notations

CAPITALS	material that must be entered exactly as it appears.
italics	words, letters, characters or values you must supply from a set of acceptable entries.
. . . (ellipsis)	items preceding the ellipsis may be repeated.
X'*nnnn*'	*nnnn* is a hexadecimal number.
O'*nnnnn*'	*nnnnn* is an octal number.
(KEYNAME)	one of the keys from your keyboard.

b̸	a blank space character (ASCII code 32). For example, in BASICb̸b̸PROG there are two spaces between BASIC and PROG.

Terms

buffer	a number between 1 and 15. This refers to an area in memory that BASIC uses to create and access a disk file. Once you use a buffer to create a file, you cannot use it to create or access any other files; you must first close the file. You may only access an open file with the buffer used to open it.
[parameters]	information you supply to specify how a command is to operate. Parameters enclosed in brackets are optional.
[expressions]	values you supply for a function to evaluate. Expressions enclosed in brackets are optional.
syntax	a command with its parameter(s), or a function with its argument(s). This shows the format to use for entering a keyword in a program line.

Terms Used in Chapter 7 for Brevity:

line	a numeric value that identifies a BASIC program line. Each line has a number between 0 and 65529.
integer	any integer expression. It may consist of an integer, or several integers joined by operators. Integers are whole numbers between –32768 and 32767.
string	any string expression. It may consist of a string, or several strings joined by operators. A string is a sequence of characters which is to be taken verbatim.
number	any numeric expression. It may consist of a number, or several numbers joined by operators.
dummy number or dummy string	a number (or string) used as a parameter to meet syntactic requirements, but whose value is insignificant.

Part II is organized this way:

Section III. Operations

Chapter 1.	Sample Session
Chapter 2.	Command Mode Execution Mode
Chapter 3.	Line Edit Mode

Section IV. The BASIC Language

Chapter 4.	BASIC Concepts
Chapter 5.	Disk Files
Chapter 6.	Introduction to BASIC Statements and Functions
Chapter 7.	BASIC Statements and Functions

Section III/ Operations

Chapter 1/ Sample Session

The easiest way to learn how BASIC operates is to write and run a program. This chapter provides sample statements and instructions to help familiarize you with the way BASIC works.

The main steps in running a program are:

A) Loading BASIC
B) Typing the program
C) Editing the program
D) Running the program
E) Saving the program on disk
F) Loading the program back into memory

Loading BASIC

After you power up your system and install the diskette, the TRSDOS Version 6 start-up logo is displayed. Then, the following prompt appears: Date?

To answer this prompt, type today's date in this format: *dd/mm/yy*; then press (ENTER). For example, for December 1, 1984, type:

```
12/01/84 (ENTER)
```

The computer converts these numbers to: Sat, Dec 1, 1984 and displays the message "TRSDOS Ready". This indicates that you are at the operating system level. To load BASIC into the system, type:

```
BASIC (ENTER)
```

A paragraph with copyright information appears on your screen, followed by: Ready

You may now begin using BASIC.

Options for Loading BASIC

When loading BASIC, you can also specify a set of options. They are:

BASIC [*program*] ([F=*number of files*][,M=*highest memory location*])

Program specifies a program to run immediately after BASIC is started.

F= specifies the maximum number of data files that may be open at any one time (from 0-15). If you omit this option, the number of files defaults to three. Each file you specify uses 564 bytes of memory.

M= specifies the highest memory location for BASIC to use. Omit this option unless you are going to call assembly-language subroutines. (In that case, you may want to set the amount of memory well below the high-memory modules of TRSDOS.) If you omit this option, the system allocates all memory up to the HIGH$ marker to BASIC. HIGH$ can be adjusted through the MEMORY library command. See the BASIC Memory Map in Appendix J for more details.

Examples

```
TRSDOS Ready
BASIC PAYROLL (F=5) (ENTER)
```

initializes BASIC, then loads and runs the program PAYROLL; allows five data files to be open; uses all memory available.

```
TRSDOS Ready
BASIC (M=45056) (ENTER)
```

initializes BASIC; allows three data files to be open; sets the highest memory location to be used by BASIC at 45056.

```
TRSDOS Ready
BASIC (M=63488,F=6) (ENTER)
```

initializes BASIC; sets the highest memory location at 63488; allows six data files to be open. Notice that the sequence in which the M= and F= options are specified is irrelevant.

```
TRSDOS Ready
BASIC (ENTER)
```

initializes BASIC; allows three data files to be open; uses all memory available.

Typing the Program

Let's write a small BASIC program. Before pressing (ENTER) after each line, check the spelling. If you have made any mistakes, use (BACKSPACE) or (←) to correct them.

```
10 A$="WILLIAM SHAKESPEARE WROTE " (ENTER)
15 B$="THE MERCHANT OF VENICE" (ENTER)
20 PRINT A$; B$ (ENTER)
```

Check your spelling again. If it is still not perfect, enter the line number where you made the mistake. Then type the entire line again.

For example, suppose you had typed:

```
15 B$="THE VERCHANT OF VENICE"
```

To correct line 15, re-type it:

```
15 B$="THE MERCHANT OF VENICE" (ENTER)
```

Then type:

```
RUN (ENTER)
```

Your screen should display:

```
WILLIAM SHAKESPEARE WROTE THE MERCHANT OF
VENICE
```

BASIC replaced line 15 in the original program with the most recent line 15.

NOTE: BASIC "reads" your program lines in numerical order. It doesn't matter if you entered line 15 after line 20; it will still read and execute 15 before "looking" at 20.

BASIC has a powerful set of commands which allow you to correct mistakes without having to re-type the entire line. These commands are discussed in Chapter 3, the "Line Edit Mode."

Saving the Program on Disk

You can save any of your BASIC programs on disk. The disk must be write-enabled and formatted. To do this, you assign it a "filespec".

For example, if you wanted to save the program we just wrote, you could assign it the filespec "AUTHOR". Type the following command:

```
SAVE "AUTHOR" (ENTER)
```

It takes a few seconds for the computer to find a place on disk to store our program. When this process is completed, it displays Ready. The program is now saved on disk.

NOTE: A filespec can have a maximum of eight alphanumeric characters. It can also have an optional extension, up to three characters long. A slash / must be included between the filespec and the extension. The first character of both the filespec and the extension must be a letter. See Section I, "Disk Files" for additional information.

Example

```
SAVE "AUTHOR/WIL" (ENTER)
```

You may also add a drive number to your filespec by typing a colon : and the drive number.

Example

```
SAVE "AUTHOR:1" (ENTER)
```

tells the computer to save "AUTHOR" on the disk in Drive 1. Otherwise, the computer assumes you to save it on the first available drive. If you do specify a disk drive, make sure you have a disk in that drive.

Loading the Program

If, after writing or running other programs, you wanted to go back and use this program again, you must "load" it back into memory. To do this, type: LOAD "filespec", R

Example

```
LOAD "AUTHOR", R (ENTER)
```

tells the computer to load the program "AUTHOR" from disk into memory; option R tells the computer to run it.

Another way to load and run a program is to type: RUN "filespec". RUN automatically loads and runs the program specified by "filespec".

The SAVE, LOAD and RUN commands are discussed in more detail in Chapter 7.

Chapter 2/ Command And Execution Modes

This chapter describes BASIC's command and execution modes. The command mode is for typing in program lines and immediate lines. The execution mode is for executing programs and immediate lines.

Command Mode

Whenever you enter the command mode, BASIC displays the prompt:

```
Ready
```

In the command mode, BASIC does not "read" your input until you complete a "logical line" by pressing (ENTER). This is called "line input", as opposed to "character input".

A logical line is a string of up to 255 characters and is always terminated by pressing (ENTER). Of these 255 characters, 249 are reserved for the line itself; the other six are reserved for the line number and the space following the line number.

A physical line, on the other hand, is one line on the display. It contains a maximum of 80 characters.

For example, if you type 100 R's and then press (ENTER), you have two physical lines, but only one logical line.

Interpretation of a Line

BASIC always ignores leading spaces in the line — it jumps ahead to the first non-space character. If this character is not a digit, BASIC treats the line as an immediate line. If it is a digit, BASIC treats the line as a program line.

For example, if you type:

```
PRINT "THE TIME IS" TIME$ (ENTER)
```

BASIC takes this as an immediate line.

But if you type:

```
10 PRINT "THE TIME IS" TIME$ (ENTER)
```

BASIC takes this as a program line.

Immediate Lines

An immediate line consists of one or more statements separated by colons. The line is executed as soon as you press (ENTER). For example:

```
Ready
CLS: PRINT "THE SQUARE ROOT OF 2 IS"; SQR(2)
```

is an immediate line. When you press (ENTER), BASIC executes it.

Program Lines

A program line consists of a line number in the range 0 to 65529, followed by one or more statements separated by colons. When you press (ENTER), the line is stored in memory, along with any other lines you have entered this way. The program is not executed until you type RUN or another execute command. For example:

```
100 CLS: PRINT "THE SQUARE ROOT OF 2 IS" SQR(2)
```

is a program line. When you press (ENTER), BASIC stores it in memory. To execute it, type:

```
RUN (ENTER)
```

NOTE: If you include numeric constants in a line, BASIC evaluates them as soon as you press (ENTER); it does not wait until you RUN the program. If any numbers are out of range for their type, BASIC returns an error message immediately after pressing (ENTER).

Special Keys in the Command Mode

(BACKSPACE) or (←) or (CTRL)(H)	Backspaces the cursor, erasing the preceding character in the line. Use this to correct typing errors before pressing (ENTER).
(SPACEBAR)	Enters a space character and advances the cursor.
(BREAK)	Interrupts line entry and starts over with a new line.
(↓) or (CTRL)(J)	Line feed — starts a new physical line without ending the current logical line.
(CAPS)	Switches the keys pressed to either all uppercase or uppercase/lowercase mode.
(ENTER)	Ends the current logical line. BASIC "takes" the line.
(SHIFT) (BACKSPACE) or (SHIFT) (←)	Deletes the current line.

Execution Mode

When BASIC is executing statements (immediate lines or programs), it is in the execution mode. In this mode, the contents of the video display are under program control.

Special Keys in the Execution Mode

(SHIFT)(@)	Pauses execution. Press any other key (except (BREAK)) to continue.

BREAK	Terminates execution and returns you to command mode.
ENTER	Interprets data entered from the keyboard as a response to the INPUT statement.
BACKSPACE, ←, CTRL H, SHIFT BACKSPACE, SHIFT ←, CAPS	These keys or key combinations have special meaning in Execution Mode when responding to INPUT and LINE INPUT statements. See the description of keys in the section "Special Keys in the Command Mode."

Chapter 3/ Line Edit Mode

This mode enables you to "debug" (correct) programs quickly and efficiently. It allows you to correct a program line without having to re-type the entire line.

If your computer encounters a syntax error while executing a program, it automatically puts you in the "line edit mode." The display shows:

```
Syntax error in line
Ready
line
```

(*line* is the program line in which the error occurred.) In this case, you are ready to use the edit mode commands and subcommands described later in this chapter.

However, if you wish to activate the line editor yourself (because you have noticed a mistake or wish to make a change in a long program line), type:

```
EDIT line (ENTER)
```

This lets you edit the specified line number. (If the line number you specify has not been used, an "Undefined line number" error occurs. If you do not have a space after the word EDIT, a "Syntax error" occurs.)

You may also type:

```
EDIT . (ENTER)
```

The period after EDIT means that you want to edit the current program line, the last line entered, the last line altered, or a line in which an error has occurred. Notice that you need to type a blank before the period; otherwise, BASIC gives you a "Syntax error" message.

For example, type the following line and press (ENTER). (To type the exponent sign ^, press (CLEAR)(;)).

```
100 FOR I = 1 TO 10 STEP .5: PRINT I, I^2, I^3:
    NEXT
```

This line will be used in exercising all the edit subcommands described below.

Now type EDIT 100 and press (ENTER). The computer displays:

```
100
```

This starts the editor. You may now begin editing line 100.

Special Keys in the Edit Mode

(ENTER)

Pressing (ENTER) in the edit mode records all the changes you made in the current line and returns you to the command mode.

(SPACEBAR)

Pressing (SPACEBAR) moves the cursor over one space to the right and displays any character stored in the preceding position. For example, using Line 100 entered above, put the computer in the edit mode so the display shows:

```
100
```

Now press (SPACEBAR). The cursor moves over one space and the first character of the program line is displayed. If this character was a blank, then a blank is displayed. Press (SPACEBAR) again until you reach the first non-blank character:

```
100 F
```

is displayed. To move over more than one space at a time, type the desired number of spaces first, then press (SPACEBAR). For example, type 6 and press (SPACEBAR). The display should show something like this (depending on how many blanks you inserted in the line):

```
100 FOR I =
```

Now type 8 and press (SPACEBAR). The cursor moves over eight spaces to the right, and eight more characters are displayed.

```
100 FOR I = 1 TO 10
```

L (List Line)

displays the remainder of the program line (unless the computer is under one of the insert subcommands listed below). The cursor drops down to the next line of the display, reprints the current line number, and moves to the first column of the line.

For example, when the display shows

```
100
```

press L (without pressing (ENTER)). Line 100 is displayed:

```
100 FOR I = 1 TO 10 STEP .5: PRINT I, I^2, I^3:
    NEXT 100
```

This lets you look at the line in its current form while you're doing the editing.

Insert Subcommand Mode

The insert subcommand mode allows you to add material to a line while editing it. The three keys you can use to enter this subcommand mode are X, I and H.

X (Extend Line)

Displays the rest of the current line. Typing (X) also moves the cursor to the end of the line and puts the computer in the insert subcommand mode. This enables you to add material to the end of the line.

For example, using Line 100, when the display shows

```
100
```

press (X) (without pressing (ENTER)) and the entire line is displayed; notice that the cursor now follows the last character on the line:

```
100 FOR I = 1 TO 10 STEP .5: PRINT I, I^2, I^3:
    NEXT
```

We can now add another statement to the line, or delete material from the line by using (BACKSPACE) or (←). For example, type

```
: PRINT "DONE"
(ENTER)
```

at the end of the line. If you typed:

```
LIST 100
```

the display should show something like this:

```
100 FOR I = 1 TO 10 STEP .5: PRINT I, I^2, I^3:
    NEXT: PRINT "DONE"
```

NOTE: If you want to continue editing the line, press (SHIFT)(↑) to get out of the insert subcommand mode.

I (Insert)

Inserts material beginning at the current cursor position on the line.

For example, type

```
EDIT 100
(ENTER)
```

then use (SPACEBAR) to move over to the decimal point in line 100. The display shows:

```
100 FOR I = 1 TO 10 STEP .
```

Suppose you want to change the increment from .5 to .25. Press the (I) key (don't press (ENTER)). The computer lets you insert material at

the current position. Type 2 now, and the display shows:

```
100 FOR I = 1 TO 10 STEP .2
```

You have made the necessary change, so press (SHIFT)(↑) to escape from the insert subcommand. Now press (L) to display the remainder of the line and move the cursor back to the beginning of the line:

```
100 FOR I = 1 TO 10 STEP .25: PRINT I, I^2, I^3:
    NEXT: PRINT "DONE"
100
```

NOTE: You can also exit the insert subcommand and save all changes by pressing (ENTER). This returns you to command mode.

H (Hack and Insert)

Deletes the remainder of a line and lets you insert material at the current cursor position.

For example, using line 100, enter the edit mode and space over until just before the PRINT "DONE" statement. Suppose you wanted to delete this statement and insert an END statement. The display shows:

```
100 FOR I = 1 TO 10 STEP .25: PRINT I, I^2, I^3:
    NEXT:
```

Press (H), then type END and press (ENTER). List the line:

```
100 FOR I = 1 TO 10 STEP .25: PRINT I, I^2, I^3:
    NEXT: END
```

should be displayed.

NOTE: To continue editing the line, press (SHIFT)(↑) to get out of the insert subcommand mode.

Other Edit Commands

A (Cancel and Restart)

Moves the cursor back to the beginning of the program line and cancels editing changes made since the last time you pressed (ENTER).

For example, if you have added, deleted, or changed something in a line, and you wish to go back to the beginning of the line and cancel the changes already made: first press (SHIFT)(↑) (to escape from any subcommand you may be executing); then press (A). The cursor drops down to the next line, displays the line number and moves to the first character position.

E (Save Changes and Exit)

Ends editing and saves all changes made. You must be in edit mode, not executing any subcommand, when you press (E) to end editing.

Q (Cancel and Exit)

Ends editing and cancels all changes made in the current editing session. If you've decided not to change the line, type (Q) to cancel changes and leave the edit mode.

If a syntax error is detected during program execution, BASIC starts the editor. To examine variable values, you must press Q before typing any other command.

*n*D (Delete)

Deletes the specified number of characters to the right of the cursor. The deleted characters appear enclosed in backslashes.

For example, using line 100, space over to just before the PRINT statement:

```
100 FOR I = 1 TO 10 STEP .25:
```

Now type 19D. This tells the computer to delete 19 characters to the right of the cursor. The display should show something like this:

```
100 FOR I = 1 TO 10 STEP .25: \PRINT I, I^2,
    I^3:\
```

When you list the complete line, you will see that everything from the PRINT to the next statement has been deleted.

*n*C (Change)

Lets you change the specified number of characters beginning at the current cursor position. If you type C without a preceding number, the computer assumes you want to change one character. When you have entered n number of characters, the computer returns you to the edit mode (so you're not in the *n*C subcommand).

For example, using line 100, suppose you want to change the final value of the FOR NEXT loop, from "10" to "15". In the edit mode, space over to just before the "0" in "10".

```
100 FOR I = 1 TO 1
```

Now press (C). The computer assumes you want to change just one character. Press (5), then press (L). When you list the line, you will see that the change has been made.

```
100 FOR I = 1 TO 15 STEP .25: NEXT: END
```

would be the current line if you've followed the editing sequence in this chapter.

*n*S*c* (Search)

Searches for the *n*th occurrence of the character *c*, and moves the cursor to that position. If you don't specify a value for *n*, the computer searches for the first occurrence of the specified character. If character *c* is not found, cursor goes to the end of the line.

NOTE: The computer only searches through characters to the right of the cursor.

For example, using the current form of line 100 type EDIT 100 (ENTER), then press (2)(S)(:). This tells the computer to search for the second occurrence of the colon character. The display should show:

```
100 FOR I = 1 TO 15 STEP .25: NEXT
```

You may now execute one of the subcommands beginning at the current cursor position. For example, suppose you want to add the counter variable after the NEXT statement. Type I to enter the insert subcommand, then type the variable name, I. That's all you want to insert, so press (SHIFT)(↑) to escape from the insert subcommand mode. The next time you list the line, it should appear as:

```
100 FOR I = 1 TO 15 STEP .25: NEXT I: END
```

*n*K*c* (Search and "Kill")

Deletes all characters up to the *n*th occurrence of character *c*, and moves the cursor to that position.

For example, using the current version of line 100, suppose we wanted to delete the entire line up to the END statement. Type EDIT 100 (ENTER), then type (2)(K)(:). This tells the computer to delete all characters up to the 2nd occurrence of the colon.

```
100 \FOR I = 1 TO 15 STEP .25: NEXT I\
```

should be displayed. The second colon still needs to be deleted, so type D. The display now shows:

```
100 \FOR I = 1 TO 15 STEP .25: NEXT I\\:\
```

Press (ENTER) and type LIST 100 (ENTER)

Line 100 should look something like this:

```
100 END
```

n(BACKSPACE)
or *n* (←)

Moves the cursor to the left by *n* spaces. If no number *n* is given, the cursor moves back one space. When the cursor backspaces, all characters in its path are erased from the display, but they are not deleted from the program. Use the space bar to advance the cursor forward and re-display the erased characters.

Section IV/ The BASIC Language

Chapter 4/ BASIC Concepts

This chapter explains how to use the full power of BASIC. This information can help programmers build powerful and efficient programs. If you are still something of a novice, you might want to skip this chapter for now, keeping in mind that the information is here when you need it.

The chapter is divided into four sections:

A. Overview — Elements of a Program. This section defines many of the terms we will be using in the chapter.

B. How BASIC Handles Data. Here we discuss how BASIC classifies and stores data. This shows you how to get BASIC to store your data in its most efficient format.

C. How BASIC Manipulates Data. This gives you an overview of all the different operators and functions you can use to manipulate and test your data.

D. How to Construct an Expression. This topic can help you in constructing powerful statements instead of using many short ones.

A- Overview: Elements of a Program

This overview defines the elements of a program.

A program is made up of "statements"; statements may have several "expressions."

We will refer to these terms during the rest of this chapter.

Program

A program is made up of one or more numbered lines. Each line contains one or more BASIC statements. BASIC allows line numbers from 0 to 65529 inclusive. You can type a maximum of 249 characters per line. BASIC reserves six other characters for the line number and for the space following the line number. You may also have two or more statements to a line, separated by colons.

Here is a sample program:

Line number | BASIC statement | Colon between statements | BASIC statement

```
100 CLS: PRINT "NORMAL MODE..."
110 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
120 FOR I = 1 TO 1000: NEXT I
130 CLS: PRINT CHR$(23); "DOUBLE-SIZE MODE..."
140 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
150 END
```

When BASIC executes a program, it handles the statements one at a time, starting with the first and proceeding to the last. Some statements, such as GOTO, ON . . . GOTO, GOSUB, change this sequence.

Statements

A statement is a complex instruction to BASIC, telling the computer to perform specific operations. For example:

```
GOTO 100
```

tells the computer to perform the operations of (1) locating line 100, (2) transferring control to that line and (3) executing the statement(s) on that line.

```
END
```

tells the computer to perform the operation of ending execution of the program.

Many statements instruct the computer to perform operations with data. For example, in the statement:

```
PRINT"SEPTEMBER REPORT"
```

the data is SEPTEMBER REPORT. The statement instructs the computer to print the data inside quotes.

Expressions

An expression is actually a general term for data. There are four types of expressions:

1. Numeric expressions, which are composed of numeric data.

Examples:

```
(1 + 5.2)/3
D
5*B
3.7682
ABS(X) + RND(0)
SIN(3 + E)
```

2. String expressions, which are composed of character data.

Examples:

```
A$
"STRING"
"STRING" + "DATA"
MO$ + "DATA"
MID$(A$,2,5) + MID$("MAN",1,2)
M$ + A$ + B$
```

3. Relational expressions, which test the relationship between two expressions.

Examples:

```
A=1
A$>B$
```

4. Logical expressions, which test the logical relationship between two expressions.

Examples:

```
A$="YES" AND B$="NO"
C>5 OR M<B OR 0>-2
578 AND 452
```

Functions

Functions are automatic subroutines. Most BASIC functions perform computations on data. Some serve a special purpose, such as controlling the video display or providing data on the status of the computer. You may use functions in the same manner that you use any data: as part of a statement.

These are some of BASIC's functions:

```
INT
ABS
STRING$
```

For example, ABS returns the absolute value of a numeric expression. The following example shows how this function works:

```
PRINT ABS(7*(-5)) (ENTER)
35
READY
```

B- How BASIC Handles Data

BASIC offers several different methods of handling your data. Using these methods properly can greatly improve the efficiency of your program. In this section we discuss:

Ways of Representing Data
Constants
Variables
How BASIC Stores Data
Numeric (integer, single precision, double precision)
String
How BASIC Classifies Constants
How BASIC Classifies Variables
How BASIC Converts Data

Ways of Representing Data

BASIC recognizes data in two forms: directly (as constants), or by reference to a memory location (as variables).

Constants

All data is input into a program as "constants" — values which are not subject to change. For example, the statement:

```
PRINT "1 PLUS 1 EQUALS"; 2
```

contains one string constant (1 PLUS 1 EQUALS), and one numeric constant (2).

In these examples, the constants "input" to the PRINT statement. They tell PRINT what data to print on the display.

These are more examples of constants:

3.14159	"L.O.SMITH"
1.775E+3	"0123456789ABCDEF"
"NAME TITLE"	−123.45E−8
57	"AGE"

Variables

A variable is a place in memory where data is stored. Unlike a constant, a variable's value can change. This allows you to write programs dealing with changing quantities. For example, in the statement:

```
A$ = "OCCUPATION"
```

The variable A$ now contains the data OCCUPATION. However, if this statement appeared later in the program:

```
A$ = "FINANCE"
```

The variable A$ would no longer contain OCCUPATION. It would now contain the data FINANCE.

Variables can also store numeric values. For example:

```
A = 134
```

Variable Names

In BASIC, variables are represented by names. Variable names must begin with a letter, A through Z. This letter may be followed by one or more characters (digits or letters).

For example:

AM A A1 BALANCE EMPLOYEE2

are all valid and distinct variable names.

Variable names may be up to 40 characters long. All characters are significant in BASIC.

Reserved Words

Certain combinations of letters are reserved as BASIC keywords and operator names. These combinations cannot be used as variable names. For example:

OR LEN OPTION

cannot be used as variable names. However, they may be embedded in a variable name. For example, OPTIONS is a valid variable name.

BASIC requires that all reserved words be delimited. This means that you must leave a blank space between a reserved word and any variables, constants or other reserved words. See Appendix F for a list of BASIC's reserved words.

Simple and Subscripted Variables

Variables may also be "subscripted" so that an entire list of data can be stored under one variable name. This method of data storage is called an *array*. For example, an array named A may contain these elements (subscripted variables):

A(0) A(1) A(2) A(3) A(4)

You may use each of these elements to store a separate data item, such as:

```
A(0) = 5.3
A(1) = 7.2
A(2) = 8.3
A(3) = 6.8
A(4) = 3.7
```

In this example, array A is a one-dimensional array, since each element contains only one subscript. An array may also be two-dimensional, with each element containing two subscripts. For example, a two-dimensional array named X could contain these elements:

```
X(0,0) = 8.6      X(0,1) = 3.5
X(1,0) = 7.3      X(1,1) = 32.6
```

With BASIC, you may have as many dimensions in your array as your program space allows. Here is an example of a three-dimensional array named L which contains these eight elements:

```
L(0,0,0) = 35233  L(0,1,0) = 96522
L(0,0,1) = 52000  L(0,1,1) = 10255

L(1,0,0) = 33333  L(1,1,0) = 96253
L(1,0,1) = 53853  L(1,1,1) = 79654
```

BASIC assumes that all arrays contain 11 elements in each dimension. If you want more elements you must use the DIM statement at the beginning of your program to dimension the array.

For example, to dimension array L, put this line at the beginning of the program:

```
DIM L(1,1,1)
```

to allow room for two elements in the first dimension; two in the second, and two in the third for a total of 2 * 2 * 2 = 8 elements.

How BASIC Stores Data

The way BASIC stores data determines the amount of memory it consumes and the speed in which BASIC can process it.

Numeric Data

You may get BASIC to store all numbers in your program as either integer, single precision, or double precision. In deciding how to get BASIC to store your numeric data, remember the trade-offs. Integers are the most efficient and the least precise. Double precision is the most precise and least efficient.

Integers
(Fastest in Computations, Limited in Range)

To be stored as an integer, a number must be whole and in the range of −32768 to 32767. An integer value requires two bytes of memory for storage. Arithmetic operations are faster when both operands are integers.

For example:

```
1      3200      -2      500      -12345
```

can all be stored as integers.

Single Precision
(General Purpose, Full Numeric Range)

Single-precision numbers can include up to seven significant digits, and can represent normalized values with exponents up to 38, i.e., numbers in the range:

$$[-1 \times 10^{38}, -1 \times 10^{-38}]\ [1 \times 10^{38}, 1 \times 10^{-38}]$$

NOTE: In this manual, a normalized value is one in which exactly one digit appears to the left of the decimal point. For example, 12.3 expressed in normalized form is 1.23×10.

If a number is raised to a power greater than 38, an "Overflow" error occurs. If it is raised to a power lower than −38, no errors are generated and program execution continues.

A single-precision value requires four bytes of memory for storage. BASIC assumes a number is single precision if you do not specify the level of precision.

For example:

10.001   −200034   1.774E6   6.024E−23   123.4567

can all be stored as single-precision values. But even though BASIC stores a number with up to seven digits of precision, when printing it, only six digits are shown.

NOTE: When used in a decimal number, the symbol E stands for "single-precision times 10 to the power of . . ." Therefore 6.024E−23 represents the single-precision value:

$$6.024 \times 10^{23}$$

Double Precision
(Maximum Precision, Slowest in Computations)

Double-precision numbers can include up to 16 significant digits, and can represent values in the same range as that for single-precision numbers. A double-precision value requires eight bytes of memory for storage. Arithmetic operations involving at least one double-precision number are slower than the same operations when all operands are single precision or integer.

For example:

1010234578
−8.7777651010
3.141592653589793
8.00100708D12

can all be stored as double-precision values.

NOTE: When used in a decimal number, the symbol D stands for "double precision times 10 to the power of . . ." Therefore

8.00100708D12 represents the value

$8.00100708 \times 10^{12}$

Strings

Strings (sequences of characters) are useful for storing non-numeric information such as names, addresses, or text. You may store ASCII characters, as well as any of the graphic and non-ASCII symbols, in a string. (A list of Character Codes is included in Appendix C).

For example, the data constant:

Jack Brown, Age 38

can be stored as a string of 18 characters. Each character (and blank) in the string is stored as an ASCII code, requiring one byte of storage.

BASIC would store the above string constant internally as:

Hex Code	4A	61	63	6B	20	42	72	6F	77	6E	2C	20	41	67	65	20	33	38
ASCII Character	J	a	c	k		B	r	o	w	n	,		A	g	e		3	8

A string can be up to 255 characters long. Strings with length zero are called "null" or "empty".

How BASIC Classifies Constants

When BASIC encounters a data constant in a statement, it must determine the type of the constant: string, integer, single precision, or double precision. First, we will list the rules BASIC uses to classify the constant. Then we will show you how you can override these rules, if you want a constant stored differently:

Rule 1

If the value is enclosed in double-quotes, it is a string.

For example:

"YES"
"3331 Waverly Way"
"1234567890"

are all classified as strings.

Rule 2

If the value is not in quotes, it is a number. (An exception to this rule is during data entry by an operator, and in DATA lists. See INPUT, INKEY$, and DATA.)

For example:

```
123001
1
-7.3214E + 6
```

are all numeric data.

Rule 3

Whole numbers in the range of −32768 to 32767 are integers.

For example:

```
12350
-12
10012
```

are integer constants.

Note: In a program statement, you enter a number as a constant in response to a command that calls for an integer, and if the number is out of integer range, BASIC converts the number to single or double precision. When the number is printed, it appears with a type-declaration tag at the end (# for double precision, ! for single precision).

Rule 4

If the number is not an integer and contains seven or fewer digits, it is single precision.

For example:

```
1234567
-1.23
1.3321
```

are all classified as single precision.

Rule 5

If the number contains more than seven digits, it is double precision. For example, these numbers:

```
1234567890123456
-1000000000000.1
2.777000321
```

are all classified as double precision.

Type Declaration Tags

You can override BASIC's normal typing criteria by adding the following "tags" at the end of the numeric constant:

! Makes the number single precision. For example, in the statement:

```
A = 12.345678901234!
```

BASIC classifies the constant as single precision, and shortens it to seven digits. However, if you tell BASIC to print the value of A, only six digits are printed out:

```
12.3457
```

E   Single-precision exponential format. The E indicates that the constant is to be multiplied by a specific power of 10. For example:

```
A = 1.2E5
```

stores the single-precision number 120000 in A.

Makes the number double precision. For example, in statement:

```
PRINT 3#/7
```

BASIC classifies the first constant as double precision before the division takes place.

D   Double-precision exponential format. The D indicates the constant is to be multiplied by a specified power of 10. For example, in:

```
A = 1.23456789D - 1
```

the double-precision constant has the value 0.123456789.

How BASIC Classifies Variables

When BASIC encounters a variable name in the program, it classifies it as either a string, an integer, a single-precision number, or a double-precision number.

BASIC classifies all variable names as single-precision initially. For example:

AB   AMOUNT   XY   L

are all single precision initially. If this is the first line of your program:

```
LP = 1.2
```

BASIC classifies LP as a single-precision variable.

However, you may assign different attributes to variables by using definition statements at the beginning of your program:

DEFINT - Defines variables as integer
DEFDBL - Defines variables as double-precision
DEFSTR - Defines variables as string
DEFSNG - Defines variables as single-precision. (Since BASIC classifies all variables as single precision initially

anyway, you would only need to use DEFSNG if one of the other DEF statements was used).

For example:

```
DEFSTR L
```

makes BASIC classify all variables which start with L as string variables. After this statement, the variables:

L   LP   LAST

can all hold string values only.

Type Declaration Tags

As with constants, you can always override the type of a variable name by adding a type declaration tag at the end. The four types of declaration tags for variables are:

%	Integer
!	Single precision
#	Double precision
$	String

For example:

I%   FT%   NUM%   COUNTER%

are all integer variables, **regardless** of what attributes have been assigned to the letters I, F, N, and C.

T!   RY!   QUAN!   PERCENT!

are all single-precision variables, **regardless** of what attributes have been assigned to the letters T, R, Q, and P.

X#   RR#   PREV#   LSTNUM#

are all double-precision variables, **regardless** of what attributes have been assigned to the letters X, R, P, and L.

Q$ CA$ WRD$ ENTRY$

are all string variables, **regardless** of what attributes have been assigned to the letters Q, C, W, and E.

Note that any given variable name can represent four different variables. For example:

A5#   A5!   A5%   A5$

are all valid and **distinct** variable names.

One further implication of type declaration: Any variable name used without a tag is equivalent to the same variable name used with one of the four tags. For example, after the statement:

```
DEFSTR C
```

the variable referenced by the name C1 is identical to the variable referenced by the name C1$.

How BASIC Converts Numeric Data

Often your program might ask BASIC to assign one type of constant to a different type of variable. For example:

```
A% = 2.34
```

In this example, BASIC must first convert the single-precision constant 2.34 to an integer in order to assign it to the integer variable A%.

You might also want to convert one type of variable to a different type, such as:

```
A# = A%
A! = A#
A! = A%
```

The conversion procedures are explained on the following pages.

Single or double precision to integer type

BASIC rounds the fractional portion of the number.

NOTE: The original value must be greater than or equal to −32768, and less than 32768.

Examples

```
A% = 32766.7
```

assigns A% the value 32767.

```
A% = 2.5D3
```

assigns A% the value 2500.

```
A% = -123.45678901234578
```

assigns A% the value −123.

```
A% = -32768.5
```

produces an Overflow Error (out of integer range).

Integer to single or double precision

No error is introduced. The converted value looks like the original value with zeros to the right of the decimal place.

Examples

```
A# = 32767
```

Stores 32767.000000000000 in A#.

```
A! = -1234
```

Stores −1234.000 in A.

Double to single precision

This involves converting a number with up to 16 significant digits into a number with no more than seven digits. BASIC rounds the number to seven significant digits. Before printing it, BASIC rounds it off to six digits.

Examples

```
A! = 1.2345678901234567
```

stores 1.234568 in A!. However, the statement:

```
PRINT A
```

displays the value 1.23457, because only six digits are displayed. The full seven digits are stored in memory.

```
A! = 1.3333333333333333
```

stores 1.333333 in A!.

Single to double precision

To make this conversion, BASIC simply adds trailing zeros to the single-precision number. If the original value has an exact binary representation in single-precision format, no error is introduced. For example:

```
A# = 1.5
```

stores 1.5000000000000 in A#, since 1.5 does have an exact binary representation.

However, for numbers which have no exact binary representation, an error is introduced when zeros are added. For example:

```
A# = 1.3
```

stores 1.299999952316284 in A#.

Because most fractional numbers do not have an exact binary representation, you should keep such conversions out of your programs. For example, whenever you assign a constant value to a double-precision variable, you can force the constant to be double precision:

```
A# = 1.3#        A# = 1.3D
```

both store 1.3 in A#.

Here is a special technique for converting a single precision value to double precision, without introducing an error into the double-

precision value. It is useful when the single-precision value is stored in a variable.

Take the single-precision variable, convert it to a string with STR$, then convert the resultant string back into a number with VAL. That is, use:

VAL(STR$(single-precision variable))

For example, the following program:

```
10 A! = 1.3
20 A# = A!
30 PRINT A#
```

prints a value of:

```
1.299999952316284
```

Compare with this program:

```
10 A! = 1.3
20 A# = VAL(STR$(A!))
30 PRINT A#
```

which prints a value of:

```
1.3
```

The conversion in line 20 causes the value in A to be stored accurately in double-precision variable A#.

Illegal Conversions

BASIC cannot automatically convert numeric values to string, or vice versa. For example, the statements:

```
A$ = 1234
A% = "1234"
```

are illegal. They would return a "Type mismatch" error. (Use STR$ and VAL to accomplish such conversions.)

C- How BASIC Manipulates Data

You have many fast methods you may use to get BASIC to count, sort, test, and rearrange your data. These methods fall into two categories:

1. Operators
   a. numeric
   b. string
   c. relational
   d. logical
2. Functions

Operators

An operator is the single symbol or word which signifies some action to be taken on either one or two specified values referred to as operands.

In general, an operator is used like this:

operand-1	operator	operand-2
6	+	2

The addition operator + connects or relates its two operands, 6 and 2, to produce the result 8.

Operand-1 and -2 can be expressions.

A few operations take only one operand, and are used like this:

operator	operand
–	5

The negative operator – acts on single operand 5 to produce the result negative 5.

Neither 6 + 2 nor –5 can stand alone; they must be used in statements to be meaningful to BASIC. For example:

```
A = 6 + 2
PRINT -5
```

Operators fall into four categories:

- Numeric
- String
- Relational
- Logical

based on the kinds of operands they require and the results they produce.

Numeric Operators

Numeric Operators are used in numeric expressions. Their operands must always be numeric, and the results they produce is one numeric data item.

In the description below, we use the terms integer, single-precision, and double-precision operations. Integer operations involve two-byte operands, single-precision operations involve four-byte operands, and double-precision operations involve eight-byte operands. The more bytes involved, the slower the operation.

There are seven different numeric operators. Two of them, sign + and sign –, are unary, that is, they have only one operand. A sign operator has no effect on the precision of its operand.

For example, in the statement:

```
PRINT -77, +77
```

the sign operators − and + produce the values negative 77 and positive 77, respectively.

NOTE: When no sign operator appears in front of a numeric term, + is assumed.

The other numeric operators are all binary, that is, they all take two operands.

These operators are, in order of precedence:

^	Exponentiation
*, /	Multiplication, Division
\, MOD	Integer Division, Modulus Arithmetic
+, −	Addition, Subtraction

Exponentiation

The symbol ^ denotes exponentiation. It converts both its operands to single precision and returns a single-precision result.

NOTE: To enter the ^ operator, press (CLEAR)(;).

For example:

```
PRINT 6^.3
```

prints 6 to the .3 power.

Multiplication

The * operator is the symbol for multiplication. Once again, BASIC uses the precision of the more precise operand to perform the operation (the less precise operand is converted).

Examples:

```
PRINT 33% * 11%
```

integer multiplication is performed.

```
PRINT 33 * 11.1
```

```
PRINT 33% * 11
```

single-precision multiplication is performed.

```
PRINT 12.345678901234567 * 11
```

double-precision multiplication is performed.

Division

The / symbol is used to indicate ordinary division. Both operands are converted to single precision or double precision, depending on their original precision:

- If either operand is double precision, then both are converted to double precision and eight-byte division is performed.
- If neither operand is double precision, then both are converted to single precision and four-byte division is performed.

Examples:

```
PRINT 3/4
```

single-precision division is performed.

```
PRINT 3.8/4
```

single-precision division is performed.

```
PRINT 3/1.2345678901234567
```

double-precision division is performed.

Integer Division

The \ (backslash) is the symbol for integer division. Both operands are rounded to integers, and the result is truncated to an integer.

Examples

```
PRINT 10 \ 4
```

prints 2.

```
PRINT 68 \ 6.99
```

prints 9.

Modulus Arithmetic

MOD is the operator for modulus arithmetic. Both operands are rounded to integers. The result is the integer that is the remainder of an integer division.

Examples

```
PRINT 10 MOD 3
```

prints 1. Ten divided by 3 is 3 with a remainder of 1.

```
PRINT 68 MOD 6.99
```

prints 5. 68 divided by 7 is 3 with a remainder of 5.

Addition

The + operator is the symbol for addition. The addition is done with the precision of the more precise operand (the less precise operand is converted).

For example, when one operand is integer type and the other is single precision, the integer is converted to single precision and four-byte addition is performed. When one operand is single precision and the other is double precision, the single-precision number is converted to double precision and eight-byte addition is performed.

Examples:

```
PRINT 2 + 3
```

integer addition is performed.

```
PRINT 3.1 + 3
```

single-precision addition is performed.

```
PRINT 1.2345678901234567 + 1
```

double-precision addition is performed.

Subtraction

The − operator is the symbol for subtraction. As with addition, the operation is done with the precision of the more precise operand (the less precise operand is converted).

Examples:

```
PRINT 33 - 11
```

integer subtraction is performed.

```
PRINT 33 - 11.1
```

single-precision subtraction is performed.

```
PRINT 12.345678901234567 - 11
```

double-precision subtraction is performed.

String Operator

BASIC has a string operator (+) which allows you to concatenate (link) two strings into one. This operator should be used as part of a string expression. The operands are both strings and the resulting value is one piece of string data.

The + operator links the string on the right of the sign to the string on the left. For example:

```
PRINT "CATS" + "LOVE" + "MICE"
```

prints:

```
CATSLOVEMICE
```

Since BASIC does not allow a string to be longer than 255 characters, you will get an error if your resulting string is too long.

Relational Operators

Relational operators compare two numerical or two string expressions to form a relational expression. This expression reports whether the comparison you set up in your program is true or false. It returns a −1 if the relation is true; a 0 if it is false.

Numeric Relations

This is the meaning of the operators when you use them to compare numeric expressions:

<	Less than
>	Greater than
=	Equal to
< > or > <	Not equal to
= < or < =	Less than or equal to
= > or > =	Greater than or equal to

Examples of true relational expressions:

```
1  <  2
2  <> 5
2  <= 5
2  <= 2
5  >  2
7  =  7
```

String Relations

The relational operators for string expressions are the same as above, although their meanings are slightly different. Instead of comparing numerical magnitudes, the operators compare their ASCII sequence. This allows you to sort string data:

<	Precedes
>	Follows
> < or < >	Does not have the same precedence
< =	Precedes or has the same precedence
> =	Follows or has the same precedence

BASIC compares the string expressions on a character-by-character basis. When it finds a non-matching character, it checks to see which character has the lower ASCII code. The character with the lower ASCII code is the smaller (precedent) of the two strings.

NOTE: Appendix C contains a listing of ASCII codes for each character.

Examples of true relational expressions:

"A" < "B"

The ASCII code for A is decimal 65; for B it's 66.

"CODE" < "COOL"

The ASCII code for O is 79; for D it's 68.

If while making the comparison, BASIC reaches the end of one string before finding non-matching characters, the shorter string is the precedent. For example:

"TRAIL" < "TRAILER"

Leading and trailing blanks are significant. For example:

" A" < "A"

ASCII for the space character is 32; for A, it's 65.

"Z-80" < "Z-80A"

The string on the left is four characters long; the string on the right is five.

How to Use Relational Expressions

Normally, relational expressions are used as the test in an IF/THEN statement. For example:

```
IF A = 1 THEN PRINT "CORRECT"
```

BASIC tests to see if A is equal to 1. If it is, BASIC prints the message.

```
IF A$ < B$ THEN 50
```

if string A$ alphabetically precedes string B$, then the program branches to line 50.

```
IF R$ = "YES" THEN PRINT A$
```

if R$ equals YES then the message stored as A$ is printed.

However, you may also use relational expressions simply to return the true or false results of a test. For example:

```
PRINT 7 = 7
```

prints −1 since the relation tested is true.

```
PRINT "A" > "B"
```

prints 0 because the relation tested is false.

Logical Operators

Logical operators make logical comparisons. Normally, they are used in IF/THEN statements to make a logical test between two or more relations. For example:

```
IF A = 1      OR C = 2      THEN PRINT X
```

The logical operator, OR, compares the two relations A = 1 and C = 2.

Logical operators may also be used to make bit comparisons of two numeric expressions.

For this application, BASIC does a bit-by-bit comparison of the two operands, according to predefined rules for the specific operator.

NOTE: The operands are converted to integer type, stored internally as 16-bit, twos complement numbers. To understand the results of bit-by-bit comparisons, you need to keep this in mind.

The following table summarizes the action of Boolean operators in bit manipulation.

Operator	Meaning of Operation	First Operand	Second Operand	Result
AND	When both bits	1	1	1
	are 1, the re-	1	0	0
	sults will be 1.	0	1	0
	Otherwise, the	0	0	0
	result will be 0.			
OR	Result will be 1	1	1	1
	unless both bits	1	0	1
	are 0.	0	1	1
		0	0	0
NOT	Result is oppo-	1		0
	site of bit.	0		1
XOR	When one of the bits	1	1	0
	is 1, the result is	1	0	1
	1. Otherwise, the	0	1	1
	result is 0.	0	0	0
EQV	When both bits are	1	1	1
	1 or both bits	1	0	0
	are 0, the	0	1	0
	result is 1.	0	0	1
IMP	The result is 1	1	1	1
	unless the first	1	0	0
	bit is 1 and the	0	1	1
	second bit is 0.	0	0	1

Hierarchy of Operators

When your expressions have multiple operators, BASIC performs the operations according to a well-defined hierarchy so that results are always predictable.

Parentheses

When a complex expression includes parentheses, BASIC always evaluates the expressions inside the parentheses before evaluating

the rest of the expression. For example, the expression:

8 − (3 − 2)

is evaluated like this:

3 − 2 = 1
8 − 1 = 7

With nested parentheses, BASIC starts evaluating the innermost level first and works outward. For example:

4 * (2 − (3 − 4))

is evaluated like this:

3 − 4 = −1
2 − (−1) = 3
4 * 3 = 12

Order of Operations

When evaluating a sequence of operations on the same level of parentheses, BASIC uses a hierarchy to determine what operation to do first.

The two listings below show the hierarchy BASIC uses. Operators are shown in decreasing order of precedence and are executed as encountered **from left to right:**

For Numeric Operations:

( )	Parentheses
^	Exponentiation
+ −	Unary sign operations, **not** addition and subtraction
* /	Multiplication and division
\	Integer division
MOD	Modulus arithmetic
+ −	Addition and subtraction
< > = <= >= <>	Relational tests
NOT	
AND	
OR	
XOR	
EQV	
IMP	

For String Operations:

+
< > = <= >= <>

For example, in the line:

X * X + 5^2.8

BASIC finds the value of 5 to the 2.8 power. Next it multiplies X*X, and finally it adds the value of 5 to the 2.8. If you want BASIC to perform the indicated operations in a different order, you must add parentheses. For example:

```
X * (X + 5)^2.8
```

or

```
X * X + (5^2.8)
```

Here's another example:

```
IF X + Ø  OR  Y > Ø  AND  Z = 1  THEN GOTO  255
```

The relational operators = and > have the highest precedence, so BASIC performs them first, one after the next, from left to right. Then the logical operations are performed. AND has a higher precedence than OR, so BASIC performs the AND operation before OR.

If the above line looks confusing because you can't remember which operator is precedent over which, then you can use parentheses to make the sequence obvious:

```
IF X = Ø OR ((Y > Ø) AND (Z = 1)) THEN GOTO 255
```

Functions

A function is a built-in sequence of operations which BASIC performs on data. BASIC functions save you from having to write a BASIC routine, and they operate faster than a BASIC routine would.

Examples:

```
SQR (A + 6)
```

tells BASIC to compute the square root of (A + 6).

```
MID$ (A$,3,2)
```

tells BASIC to return a substring of the string A$, starting with the third character, with a length of 2.

BASIC functions are described in more detail in Chapter 7.

If the function returns numeric data, it is a numeric function and may be used in a numeric expression. If it returns string data, it is a string function and may be used in a string expression.

D- How to Construct an Expression

Understanding how to construct an expression will help you put together powerful statements — instead of using many short ones. In this section we will discuss the two kinds of expressions you may construct:

- Simple
- Complex

as well as how to construct a function.

As we have stated before, an expression is actually data. This is because once BASIC performs all the operations, it returns one data item. An expression may be string or numeric. It may be composed of:

- Constants
- Variables
- Operators
- Functions

Expressions may be either simple or complex:

A **simple expression** consists of a single term: a constant, variable or function. If it is a numeric term, it may be preceded by an optional + or − sign, or by the logical operator NOT.

For example:

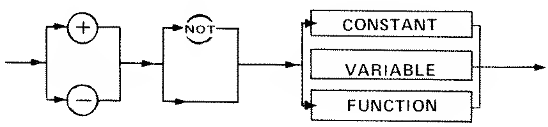
+A   3.3   −5   SQR(8)

are all simple numeric expressions, since they only consist of one numeric term.

A$ STRING$ (20,A$)   "WORD"   "M"

are all simple string expressions, since they only consist of one string term.

Here's how a **simple expression** is formed:



A *complex expression* consists of two or more terms (simple expressions) combined by operators. For example:

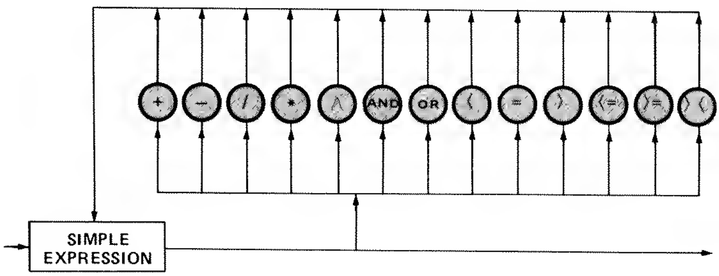
A-1   X+3.2−Y   1=1   A AND B   ABS(B)+LOG(2)

are all examples of complex numeric expressions. (Notice that you can use the relational expression (1=1) and the logical expression (A AND B) as a complex numeric expression since both actually return numeric data.)

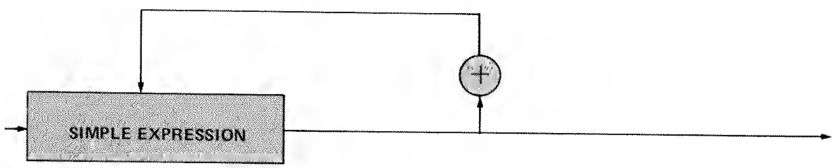
A$ + B$ "Z" + Z$ STRING$(10, "A") + "M"

are all examples of complex string expressions.

This is how a **complex numeric expression** is formed:



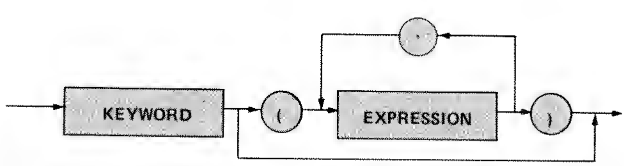
This is how a **complex string expression** is formed:



Most functions, except functions returning system information, require that you input either or both of the following kinds of data:

- One or more numeric expressions
- One or more string expressions

This is how a **function** is formed:



If the data returned is a number, the function may be used as a term in a numeric expression. If the data is a string, the function may be used as a term in a string expression.

SIN(A)   STR$(X)   VAL(A)   LOG(.53)

are all examples of functions.

Chapter 5/ Disk Files

You may want to store data on your disk for future use. To do this, you need to store the data in a "disk file." A disk file is an organized collection of related data. It may contain a mailing list, a personnel record, or almost any kind of information.

To transfer data from a BASIC program to a disk file, and vice versa, the data must first go through a "buffer". This is an area in memory where data is accumulated for further processing.

With BASIC, you can create and access two types of disk files. The difference between these two types is that each is created in a different "mode." The mode you choose determines what kind of access you will have to the file: sequential access or direct access.

Sequential-Access Files

With a sequential-access file, you can only access data in the same order it was stored: sequentially. To read from or write to a particular section in the file, you must first read through all the contents in the file until you get to the desired section.

Data is stored in a sequential file as ASCII characters. Therefore, it is ideal for storing free-form data without wasting space between data items. However, it is limited in flexibility and speed.

The statements and functions used with sequential files are:

OPEN	WRITE#	EOF
PRINT#	INPUT#	LOC
PRINT# USING	LINE INPUT#	CLOSE

These statements and functions are discussed in more detail in Chapters 6 and 7.

Creating a Sequential-Access File

1. To create the file, OPEN it in "O" (output) mode and assign it a buffer number (from 1 to 15).

   Example

   ```
   OPEN "O", 1, "LIST/EMP"
   ```

   opens a sequential output file named LIST/EMP and gives buffer 1 access to this file.

2. To input data from the keyboard into one or more program variables, use either INPUT or LINE INPUT. (The difference between these two statements is that each recognizes a different set of "delimiters". Delimiters are characters that define where a data item begins or ends).

Example

```
LINE INPUT, "NAME? "; N$
```

inputs data from the keyboard and stores it in variable N$.

3. To write data to the file, use the WRITE# statement (you can also use PRINT#, but make sure you delimit the data).

   Example

   ```
   WRITE# 1, N$
   ```

   writes variable N$ to the file, using buffer 1 (the buffer used to OPEN the file). Remember that data must go through a buffer before it can be written to a file.

4. To ensure that all the data was written to the file, use the CLOSE statement.

   Example

   ```
   CLOSE 1
   ```

   closes access to the file, using buffer 1 (the same buffer used to OPEN the file).

   Sample Program

   ```
   10 OPEN "O", 1, "LIST/EMP"
   20 LINE INPUT "NAME? ";N$
   30 IF N$ = "DONE" THEN 60
   40 WRITE# 1, N$
   50 PRINT: GOTO 20
   60 CLOSE 1
   RUN
   ```

   NOTE: The file "LIST/EMP" stores the data you input through the aid of the program, not the program itself (the program manipulates data). To save the program above, you must assign it a name using the SAVE command (refer to Chapter 1).

   Example

   ```
   SAVE "PAYROLL"
   ```

   would save the program under the name "PAYROLL".

   NOTE: Every time you modify a program, you must SAVE it again (you can use the same name); otherwise, the original program remains on disk, without your latest corrections.

5. To access data in the file, reOPEN it in the "I" (input) mode.

   Example

   ```
   OPEN "I", 1, "LIST/EMP"
   ```

OPENs the file named LIST/EMP for sequential input, using buffer 1.

6. To read data from the file and assign it to program variables, use either INPUT# or LINE INPUT#.

   Examples

   ```
   INPUT# 1, N$
   ```

   reads a string item into N$, using buffer 1 (the buffer used when the file was OPENed).

   ```
   LINE INPUT# 1, N$
   ```

   reads an entire line of data into N$, using buffer 1.

   INPUT# and LINE INPUT# each recognize a different set of "delimiters" for reading data from the file. Delimiters are characters that define the beginning or end of a data item. See Chapter 7 for a detailed explanation of these statements.

   Sample Program

   ```
   10 OPEN "I", 1, "LIST/EMP"
   20 IF EOF(1), THEN 100
   30 INPUT# 1, N$
   40 PRINT N$
   50 GOTO 20
   100 CLOSE
   ```

Updating a Sequential-Access File

1. To add data to the file, OPEN it in "E" (extend) mode.

   ```
   OPEN "E", 1, "LIST/EMP"
   ```

   opens the file LIST/EMP so that it can be extended. The data you enter is appended to LIST/EMP.

2. To enter new data to the file, follow the same procedure as for entering data in "O" mode.

   The following program illustrates this technique. It builds upon the file we previously created under the name LIST/EMP.

   NOTE: Read through the entire program first. If you encounter BASIC words (commands or functions) that are unfamiliar to you, refer to Chapter 7 for their definitions.

   ```
   NEW
   10 OPEN "E", 1, "LIST/EMP"
   20 LINE INPUT "TYPE A NEW NAME OR PRESS <N>"; N$
   30 IF N$ = "N" THEN 60
   40 WRITE# 1, N$
   50 GOTO 20
   60 CLOSE
   ```

If you want the program to print on your display the information stored in the updated file, add the following lines:

```
70 OPEN "I", 1, "LIST/EMP"
80 IF EOF(1) THEN 2000
90 INPUT# 1, N$
100 PRINT N$
110 GOTO 80
2000 CLOSE
RUN
```

Once you have RUN this program, SAVE it.

Example

```
SAVE "PAYROLL2"          'saves the new program
```

Direct-Access Files

With a direct-access file, you can access data almost anywhere on disk. It is not necessary to read through all the information, as with a sequential-access file. This is possible because in a direct-access file, information is stored and accessed in distinct units called "records". Each record is numbered.

Creating and accessing direct-access files requires more program steps than sequential-access files. However, direct-access files are more flexible and easier to update.

One important note: BASIC allocates space for records in numeric order. That is, if the first record you write to the file is number 200, BASIC allocates space for records 0 through 199 before storing record 200 in the file.

The maximum number of logical records is 65,535. Each record may contain between 1 and 256 bytes.

The statements and functions used with direct-access files are:

OPEN	FIELD	LSET/RSET
GET	PUT	CLOSE
LOC	MKD$	MKI$
MKS$	CVD	CVI
CVS		

These statements and functions are discussed in more detail in Chapters 6 and 7.

Creating a Direct-Access File

1. To create the file, OPEN it for direct access in "D" mode ("R" may also be used. It stands for "random access", which is simply another name for direct access).

Example

```
OPEN, "D", 1, "LISTING", 32
```

opens the file named "LISTING", gives buffer 1 direct access to the file, and sets the record length to 32 bytes. (If the record length is omitted, the default is 256 bytes). Remember that data is passed to and from disk in records.

2. Use the FIELD statement to allocate space in the buffer for the variables that will be written to the file. This is necessary because you must place the entire record into the buffer before putting it into the disk file.

   Example

   ```
   FIELD 1, 20 AS N$,  4 AS A$,8 AS P$
   ```

   allocates the first 20 positions in buffer 1 to string variable N$, the next four positions to A$, and the next eight positions to P$. N$, A$ and P$ are now "field names".

3. To move data into the buffer, use the LSET statement. Numeric values must be converted into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single-precision value, and MKD$ for a double-precision value.

   Example

   ```
   LSET N$=X$
   LSET A$=MKS$(AMT)
   ```

Note: RSET right justifies a string into the buffer. For example, RSET N$ = X$.

4. To write data from the buffer to a record (within a direct-access disk file), use the PUT statement.

   ```
   PUT 1, CODE%
   ```

   writes the data from buffer 1 to a record with the number CODE%. (The percentage sign at the end of a variable specifies that it is an integer variable.)

   The following program writes information to a direct-access file:

   ```
   10 OPEN "D", 1, "LISTING", 32
   20 FIELD 1, 20 AS N$, 4 AS A$, 8 AS P$
   30 INPUT "2-DIGIT CODE, 0 TO END"; CODE%
   40 IF CODE% = 0 THEN 130
   50 INPUT "NAME"; X$
   60 INPUT "AMOUNT"; AMT
   70 INPUT "PHONE"; TEL$
   80 LSET N$ = X$
   90 LSET A$ = MKS$(AMT)
   ```

```
100 LSET P$ = TEL$
110 PUT 1, CODE%
120 GOTO 30
130 CLOSE 1
```

The two-digit code that you enter in line 30 becomes a record number. That record number will store the name(s), amount(s) and phone number(s) you enter when lines 50, 60 and 70 are executed. The record is written to the file when BASIC executes the PUT statement in line 110.

After typing this program, SAVE it and RUN it. Then, enter the following data:

```
2-DIGIT CODE, 0 TO END? 20
NAME? SMITH
AMOUNT? 34.55
PHONE? 567-9000
2-DIGIT CODE, 0 TO END? 0
```

BASIC stored SMITH, 34.55, and 567-9000 in record 20 of file LISTING.

Accessing a Direct-Access File

1. OPEN the file in "D" mode ("R" can also be used).

   Example

   ```
   OPEN "D", 1,"FILE",32
   ```

2. Use the FIELD statement to allocate space in the buffer for the variables that will be read from the file.

   Example

   ```
   FIELD 1, 20 AS N$, 4 AS A$, 8 AS P$
   ```

3. Use the GET statement to read the desired record from a direct disk file into a buffer.

   Example

   ```
   GET 1, CODE%
   ```

   gets the record numbered CODE% and reads it into buffer 1.

4. Convert string values back to numbers using the "convert" functions: CVI for integers, CVS for single-precision values, and CVD for double-precision values.

   Example

   ```
   PRINT N$
   PRINT CVS(A$)
   ```

The program may now access the data in the buffer.

The following program accesses the direct-access file "LISTING" (created with the previous program). When BASIC executes line 30, enter any *valid* record number from "LISTING". This program will print the contents of that record.

```
10 OPEN "D", 1, "LISTING", 32
20 FIELD 1,20 AS N$,4 AS A$,8 AS P$
30 INPUT "2-DIGIT CODE, 0 TO END"; CODE%
35 IF CODE% = 0 THEN 1000
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$#.##"; CVS(A$)
70 PRINT P$: PRINT
80 GOTO 30
1000 CLOSE 1
```

After typing this program, SAVE it and RUN it. When BASIC asks you to enter a 2-digit code, enter 20 (the record we created through the previous program). Your display should show:

```
2-DIGIT CODE, 0 TO END? 20
SMITH
$34.55
567-9000
```

If you entered a record number which is not a part of "LISTING", your display would show:

```
$0.00
```

If you wanted to go back and update "LISTING", simply LOAD the previous program (the one that created "LISTING") and RUN it.

Chapter 6/ Introduction To BASIC Statements And Functions

BASIC is made up of keywords. These keywords instruct the computer to perform certain operations.

Chapter 7 describes all of BASIC's keywords. This chapter explains the format used in Chapter 7. It also introduces you to BASIC's two types of keywords: statements and functions.

Format for Chapter 7

> **Keyword**
>
> **Syntax with *parameter(s)* or *(expression(s))***
>
> Brief definition of keyword.
>
> Detailed definition of keyword.
>
> Example(s)
>
> Sample Program(s)

This format varies slightly, depending on the complexity of each keyword. For instance, some keywords are used alone (without parameters or expressions). Others have several possible syntaxes. As a general rule, definitions for statements are longer than definitions for functions. That is because a statement is a complete instruction to BASIC, while a function is a built-in subroutine which may only be used as part of a statement.

Some keywords have several sample programs, others don't have any at all. We added programs to illustrate useful applications which may not be readily apparent. Remember that the examples are for reference only.

IMPORTANT NOTE: BASIC requires that keywords be delimited by spaces. This means that you must leave a space between a keyword and any variables, constants or other keywords. The only exceptions to this rule are characters which are shown as part of the syntax of the keyword.

For example, if you typed:

```
DELETE.
```

BASIC would return a "Syntax error." You must leave a blank space between the word DELETE and the period.

For a definition of the terms and notation used in Chapter 7, see page 2-4 of the Introduction.

Statements

A program is made up of lines; each line contains one or more statements. A statement tells the computer to perform some operation when that particular line is executed. For example,

```
100 STOP
```

tells the computer to stop executing the program when it reaches line 100.

Statements for assigning values to variables and defining memory space:

CLEAR	clears all variables, allocates memory and stack space.
COMMON	passes variables to a CHAINed program.
DATA	stores data in your program so that you may assign it to a variable.
DEFDBL	defines variables as double precision.
DEF FN	defines a function according to your specifications.
DEFINT	defines variables as integers.
DEFSNG	defines variables as single precision.
DEFSTR	defines variables as strings.
DEF USR	defines the entry point for USR routines.
DIM	dimensions an array.
ERASE	erases an array.
LET	assigns a value to a variable (the keyword LET may be omitted).
MID$	replaces a portion of a string.
OPTION BASE	declares the minimum value for array subscripts.
RANDOM	reseeds the random number generator.
READ	reads data stored in the DATA statement and assigns it to a variable.
RESTORE	restores the DATA pointer.
SWAP	exchanges the values of variables.

Statements for altering program sequence:

CHAIN	loads another program and passes variables to the current program.
END	ends a program.
FOR/NEXT	establishes a program loop.
GOSUB	transfers program control to the subroutine.
GOTO	transfers program control to the specified line number.
IF . . . THEN . . . ELSE	evaluates an expression and performs an operation if conditions are met.
ON . . . GOSUB	evaluates an expression and branches to a subroutine.

ON . . . GOTO	evaluates an expression and branches to another program line.
RETURN	returns from a subroutine to the calling program.
STOP	stops program execution.
WHILE . . . WEND	executes statements in a loop as long as a given condition is true.
WAIT	suspends program execution while monitoring the status of a machine input port.

Statements for storing and accessing data on disk:

CLOSE	closes access to a disk file.
FIELD	organizes a direct-access buffer.
GET	gets a record from a direct-access file.
INPUT#	inputs data from a disk file.
LINE INPUT#	inputs an entire line from a disk file.
LSET	moves data (and left-justifies it) to a field in a direct-access file buffer.
OPEN	opens a disk file.
PRINT#	writes data to a sequential disk file.
PRINT# USING	writes data to a disk file using the specified format.
PUT	puts a record into a direct-access file.
RSET	moves data (and right-justifies it) to a field in a direct-access file buffer.
WRITE#	writes data to a sequential file.

Statements for debugging a program:

CONT	continues program execution.
ERL	returns the line number where an error occurred.
ERR	returns an error code after an error.
ERROR	simulates the specified error.
ON ERROR GOTO	sets up an error-trapping routine.
RESUME	terminates an error-handling routine.
REM	inserts a remark line in a program.
TROFF	turns the tracer off.
TRON	turns the tracer on.

Statements for inputting or outputting data to the video display or the line printer:

CLS	clears the display.
INPUT	inputs data from the keyboard.
LINE INPUT	inputs an entire line from the keyboard.
LIST	lists a program to the display.
LLIST	lists program to line printer.
LPRINT	prints data at the line printer.

LPRINT USING	prints data at the line printer, using the specified format.
LPRINT TAB	moves the printer's print head to position *n* on the current line.
PRINT	prints data to the display.
PRINT USING	prints data to the display, using the specified format.
PRINT@	specifies where printing is to begin.
PRINT TAB	moves the cursor to position *n* on the current line.
WIDTH	sets number of characters to print per line on the display or line printer.
WRITE	prints data on the display.

Statements for performing system functions or entering other modes of operation:

AUTO	automatically numbers program lines.
CALL	calls an assembly-language subroutine.
DELETE	erases program lines from memory.
DEF USR	specifies the starting address of an assembly-language subroutine.
EDIT	edits program lines.
KILL	deletes a disk file.
LOAD	loads a program from disk.
MERGE	merges a disk program with a resident program.
NAME	renames a disk file.
NEW	erases a program from RAM.
OUT	sends a byte to a machine output port.
POKE	writes a byte into a memory location.
RENUM	renumbers a program.
RUN	executes a program.
SAVE	saves a program on disk.
SOUND	generates a sound
SYSTEM	returns to TRSDOS.

Functions

A function is a built-in subroutine. It may only be used as part of a statement.

Most BASIC functions return numeric or string data by performing certain built-in routines. Special print functions are used to control the video display.

Numeric Functions (return a number):

ABS	computes the absolute value.
ASC	returns the ASCII code.
ATN	computes the arctangent.
CDBL	converts to double precision.

CINT	returns the largest integer not greater than the parameter.
COS	computes the cosine.
CSNG	converts to single precision.
EXP	computes the natural exponential.
FIX	truncates to whole number.
FRE	returns the number of bytes in memory not being used.
INSTR	searches for a specified string.
INP	returns the byte read from a port.
INT	returns the largest whole number not greater than the argument.
LEN	returns the length of the string.
LOG	computes the natural logarithm.
MEM	returns the amount of memory.
PEEK	returns a byte from a memory location.
RND	returns a pseudorandom number.
SGN	returns the sign.
SIN	calculates the sign.
SQR	calculates the square root.
TAN	computes the tangent.
USR	calls an assembly-language subroutine.
VAL	returns the numeric value of a string.
VARPTR	returns an address for a variable or buffer.

String Functions (return a string value):

CHR$	returns the specified character.
DATE$	returns today's date.
ERRS$	returns the latest TRSDOS error number and message.
HEX$	converts a decimal value to a hexadecimal string.
LEFT$	returns the left portion of a string.
MID$	returns the mid-portion of a string.
OCT$	converts a decimal value to an octal string.
RIGHT$	returns the right portion of a string.
SPACE$	returns a string of spaces.
STR$	converts to string type.
STRING$	returns a string of characters.
TIME$	returns the time.

Input/Output Functions (perform input/output to the keyboard, display, line printer or disk files):

INKEY$	returns the keyboard character.
INPUT$	returns a string of characters from the keyboard.
POS	returns the cursor column position on the display.
ROW	returns the row position on the display.
SPC	prints spaces to the display.

CVD	restores data from a direct disk file to double precision.
CVI	restores data from a direct disk file to integer.
CVS	restores data from a direct disk file to single precision.
EOF	checks for end-of-file.
INPUT$	inputs a string of characters from a sequential disk file.
LOC	returns the current disk file record number.
LOF	returns the disk file's end-of-file.
LPOS	returns the logical position of the printer's print head within the printer's buffer.
MKI$	converts an integer value to a string for writing it to a direct-access disk file.
MKS$	converts a single-precision number to a string for writing it to a direct-access file.
MKD$	converts a double-precision value to a string for writing it to a direct-access file.

ABS

ABS(*number*)	**Function**

Computes the absolute value of *number*.

ABS returns the absolute value of the argument, that is, the magnitude of the number without respect to its sign.

If *number* is greater than or equal to zero, ABS(*number*) = *number*. If *number* is less than zero, ABS(*negative number*) = *number*.

Example

```
X = ABS(Y)
```

computes the absolute value of Y and assigns it to X.

Sample Program

```
100 INPUT "WHAT'S THE TEMPERATURE OUTSIDE
    (DEGREES F)"; TEMP
110 IF TEMP < 0 THEN PRINT "THAT'S" ABS(TEMP)
    "BELOW ZERO!  BRR!": END
120 IF TEMP = 0 THEN PRINT "ZERO DEGREES! MITE
    COLD!": END
130 PRINT TEMP "DEGREES ABOVE ZERO?  BALMY!":
    END
```

ASC(*string*)	**Function**

Returns the ASCII code for the first character of *string*.

The value is returned as a decimal number. If *string* is null, an "Illegal function call" error occurs.

Example

```
PRINT ASC("A")
```

prints 65, the ASCII code for "A".

Sample Programs

ASC can be used to make sure that a program is receiving the proper input. Suppose you've written a program that requires the user to input hexadecimal digits 0-9, A-F. To make sure that only those characters are input, and exclude all other characters, you can insert the following routine.

```
100 INPUT "ENTER A HEXADECIMAL VALUE
    (0-9,A-F)";N$
110 A = ASC(N$)              'get ASCII code
120 IF A>47 AND A<58 OR A>64 AND A<71 THEN PRINT
    "OK.": GOTO 100
130 PRINT "VALUE NOT OK." : GOTO 100
```

ASC can also be used to program the special function keys, as in the following program.

```
100 CLS : PRINT "Enter ANY Keyboard Character : ";
110 IN$ = INKEY$ : IF IN$ = "" /THEN GOTO 110
120 A = ASC(IN$)
130 IF A = 129 THEN IN$ = CHR$(13) + "F1 KEY" +
    CHR$(13)
140 IF A = 130 THEN IN$ = CHR$(13) + "F2 KEY" +
    CHR$(13)
150 IF A = 131 THEN IN$ = CHR$(13) + "F3 KEY" +
    CHR$(13)
160 PRINT IN$;
170 GOTO 110
180 END
```

ATN

ATN(*number*)	**Function**

Computes the arctangent of *number* in radians.

ATN returns the angle whose tangent is *number*. The result is always single precision, regardless of *number*'s numeric type.

To convert this value to degrees, multiply ATN(*number*) by 57.29578.

Example

```
X = ATN(Y/3)
```

computes the arctangent of Y/3 and assigns the value to X.

AUTO

AUTO [*line*][,*increment*]	**Statement**

Automatically generates a line number every time you press (ENTER). Immediately following the line number, you can enter your text for that line.

AUTO begins numbering at *line* and displays the next line using *increment*. The default for both values is 10. A period (.) can be substituted for *line*. In this case, BASIC uses the current line number.

If AUTO generates a line number that has already been used, it displays an asterisk after the number. To save the existing line, press (ENTER) immediately after the asterisk. AUTO then generates the next line number.

To turn off AUTO, press (BREAK). The current line is canceled and BASIC returns to command level.

Examples

```
AUTO
```

generates lines 10, 20, 30, 40.

```
AUTO 100, 50
```

generates lines 100, 150, 200, 250 . . .

CALL

CALL *variable* [*(parameter list)*]	**Statement**

Transfers program control to an assembly-language subroutine stored at *variable*.

Variable contains the address where the subroutine starts in memory. *Variable* may not be an array variable.

Parameter list contains the values that are passed to the external subroutine. *Parameter list* may contain only variables.

A CALL statement with no parameters generates a simple Z-80 "CALL" instruction. The corresponding subroutine should return with a simple "RET".

The method for passing parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers. HL contains the address pointing to parameter 1. DE contains the address pointing to parameter 2. BC contains the address pointing to parameter 3.
2. If the number of parameters is greater than 3, they are passed as follows:

   HL contains the address pointing to parameter 1.

   DE contains the address pointing to parameter 2.

   BC points to the low byte of a contiguous data block containing parameters 3 through n (that is, to the low byte of parameter 3).

Note that with this scheme, the subroutine must know how many parameters to expect in order to find them. The calling program is responsible for passing the correct number of parameters.

When accessing parameters in a subroutine, remember that they are pointers to the actual arguments passed.

NOTE: The number, type and length of the parameters in the calling program must match with the parameters expected by the subroutine. This applies to BASIC subroutines, as well as those subroutines written in assembly language.

See also USR and VARPTR.

Example

```
110 MYROUT = &HD000
120 CALL MYROUT(I,J,K)
```

We assume that D000 is the address for an assembly-language routine. The values of I, J, and K (which we also assume were given elsewhere) are passed to that routine.

CDBL

CDBL(*number*)	Function

Converts *number* to double precision.

CDBL returns a 17-digit value. This function may be useful if you want to force an operation to be performed in double precision, even though the operands are single precision or integers.

Sample Program

```
210 A=454.67
220 PRINT A; CDBL(A)
RUN
 454.67 454.6700134277344
Ready
```

CHAIN

CHAIN [**MERGE**] ***"filespec"*** [,[***line***] [,**ALL**] [,**DELETE** ***line-line***]]	**Statement**

Causes a second BASIC program, named *filespec*, to be loaded and executed.

Filespec must have been saved in ASCII format before you can CHAIN it. To do this, use SAVE with the 'A' option.

Line is the first line to be run in the CHAINed program. If omitted, execution begins at the first program line of the CHAINed program.

The ALL option passes every variable in the main program to the chained program. If omitted, the main program must contain a COMMON statement to pass variables. If you will be CHAINing subsequent programs (and passing variables), each new program must contain a COMMON statement.

The MERGE option "overlays" the lines of *filespec* with the main program. See MERGE to understand how BASIC overlays (merges) program lines.

The DELETE option deletes *lines* in the overlay so that you can MERGE in a new overlay.

Examples

```
"CHAIN PROG2"
```

loads PROG2, chains it to the main program currently in memory, and begins executing it.

```
"CHAIN SUBPROG/BAS" ,, ALL
```

loads, chains and executes SUBPROG/BAS. The values of all the variables in the main program are passed to SUBPROG/BAS.

Sample Program 1

```
10 REM THIS PROGRAM DEMONSTRATES
   CHAINING USING COMMON TO PASS VARIABLES.
20 REM SAVE THIS MODULE ON DISK AS "PROG1"
   USING THE A OPTION.
30 DIM A$(2),B$(2)
40 COMMON A$(),B$()
50 A$(1)="VARIABLES IN COMMON MUST BE ASSIGNED "
```

```
60 A$(2)="VALUES BEFORE CHAINING"
70 B$(1)="":B$(2)=""
80 CHAIN "PROG2"
90 PRINT : PRINT B$(1): PRINT : PRINT B$(2):
   PRINT
100 END
```

Save this program as "PROG1", using the 'A' option (Type: SAVE *"filespec"*, A). Type NEW, then enter the following program.

```
10 REM THE STATEMENT "DIM A$(2),B$(2)" MAY
   ONLY BE EXECUTED ONCE.
20 REM HENCE, IT DOES NOT APPEAR IN THIS MODULE.
30 REM SAVE THIS MODULE ON THE DISK AS "PROG2"
   USING THE A OPTION.
40 COMMON A$(),B$()
50 PRINT: PRINT A$(1);A$(2)
60 B$(1)="NOTE HOW THE OPTION OF SPECIFYING A
   STARTING LINE NUMBER"
70 B$(2)="WHEN CHAINING AVOIDS THE DIMENSION
   STATEMENT IN 'PROG1'."
80 CHAIN "PROG1",90
90 END
```

Save this program as "PROG2", using the 'A' option. Load PROG1 and run it. Your screen should display:

```
VARIABLES IN COMMON MUST BE ASSIGNED VALUES
BEFORE CHAINING. NOTE HOW THE OPTION OF
SPECIFYING A STARTING LINE NUMBER WHEN
CHAINING AVOIDS THE DIMENSION STATEMENT IN
'PROG1'.
```

Type NEW and this program:

Sample Program 2

```
10 REM THIS PROGRAM DEMONSTRATES CHAINING
   USING THE MERGE AND ALL OPTIONS.
20 A$="MAINPROG"
30 CHAIN MERGE "OVRLAY1", 1000, ALL
40 END
```

Save this program as "MAINPROG", using the 'A' option. Enter NEW, then type:

```
1000 PRINT A$;" HAS CHAINED TO OVRLAY1."
1010 A$="OVRLAY1"
1020 B$="OVRLAY2"
1030 CHAIN MERGE "OVRLAY2", 1000, ALL , DELETE
     1020-1040
1040 END
```

Save this program as "OVRLAY1", using the 'A' option. Enter NEW, then type:

```
1000 PRINT A$; " HAS CHAINED TO ";B$;"."
1010 END
```

Save this program as "OVRLAY2", using the 'A' option. Load MAINPROG and run it. Your screen should display:

```
MAINPROG HAS CHAINED TO OVRLAY1.
OVRLAY1 HAS CHAINED TO OVRLAY2.
```

NOTE

The CHAIN statement with the MERGE option leaves the files open and preserves the current OPTION BASE setting.

If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the chained program.

When using the MERGE option, user-defined functions should be placed before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions will be undefined after the merge is complete.

CHR$

CHR$(*code*)	**Function**

Returns the character corresponding to an ASCII or control *code*.

This is the inverse of the ASC function. CHR$ is commonly used to send a special character to the display.

Examples

```
PRINT CHR$(35)
```

prints the character corresponding to ASCII code 35 (the character is #).

```
PRINT CHR$(16)
```

puts the display into its black-on-white or black-on-green mode, also called reverse video mode. PRINT CHR$(28) returns it to white-on-black or green-on-black and converts all reverse video characters into graphics characters. See Appendix C for more information.

Sample Program

The following program lets you investigate the effect of printing codes 32 through 255 on the display. (Codes 0-31 represent certain control functions.)

```
100 CLS
110 INPUT "TYPE IN THE CODE (32-255)"; C
120 PRINT CHR$(C);
130 GOTO 110
```

For a complete list and discussion of output to the video display, see the Character Codes table in Appendix C. See also the sample program given for the ASC function of BASIC.

CINT

CINT(*number*)	**Function**

Converts *number* to integer representation.

CINT rounds the fractional portion of *number* to make it an integer.

For example, PRINT CINT(1.5) returns 2; PRINT CINT(−1.5) returns −2. The result is a two-byte integer.

Sample Program

```
PRINT CINT(17.65)
 18
Ready
```

CLEAR

CLEAR [,*memory location*] [,*stack space*]	**Statement**

Clears the value of all variables and CLOSEs all open files.

Memory location must be an integer. It specifies the highest memory location available for BASIC. The default is the current top of memory (as specified when BASIC was loaded or by the location of HIGH$). This option is useful if you will be loading a machine-language subroutine, since it prevents BASIC from using that memory area.

Stack space must also be an integer. This sets aside memory for temporarily storing internal data and addresses during subroutine calls and during FOR/NEXT loops. The default is 512 bytes or one-eighth of the memory available, whichever is smaller. An "Out of memory" error occurs if there is insufficient stack space for program execution.

NOTE: BASIC allocates string space dynamically. An "Out of string space" error occurs only if no free memory is left for BASIC.

Since CLEAR initializes all variables, you must use it near the beginning of your program, before any variables have been defined and before any DEF statements.

Examples:

```
CLEAR
```

clears all variables and closes all files.

```
CLEAR ,45000
```

clears all variables and closes all files; makes 45000 the highest address BASIC may use to run your programs.

```
CLEAR ,61000,200
```

clears all variables and closes all files; makes 61000 the highest address BASIC may use to run your programs, and allocates 200 bytes for stack space.

CLOSE

CLOSE [*buffer, . . .*]	**Statement**

Closes access to a file.

Buffer is a number from 1 - 15 used to OPEN the file. If no buffers are specified, BASIC closes all open files.

This command terminates access to a file through the specified buffer. If a *buffer* was not assigned in a previous OPEN statement, then

```
CLOSE buffer
```

has no effect.

Do not remove a diskette which contains an open file. CLOSE the file first. This is because the last records may not have been written to disk yet. Closing the file writes the data, if it hasn't already been written.

See also OPEN and the chapter on 'Disk Files'.

Examples

```
CLOSE 1, 2, 8
```

terminates the file assignments to buffers 1,2, and 8. These buffers can now be assigned to other files with OPEN statements.

```
CLOSE FIRST% + COUNT%
```

terminates the file assignment to the buffer specified by the sum FIRST% + COUNT%.

CLS

CLS	Statement

Clears the screen and moves the cursor to the upper-left corner. All characters on the screen are erased.

Reverse video is enabled, and the screen is set to 80-column mode.

Sample Program

```
540 CLS
550 FOR I = 1 TO 24
560 PRINT STRING$(79,33)
570 NEXT I
580 GOTO 540
```

COMMON

COMMON *variable, . . .*	Statement

Reserves space for *variables* so they can be passed to a CHAINed program.

COMMON may appear anywhere in a program, but we recommend using it at the beginning.

The same variable cannot appear in more than one COMMON statement. To specify array variables, append "( )" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

NOTE: array variables used in a COMMON statement must have been declared in a DIM statement.

Example

```
90 DIM D(50)
100 COMMON A, B, C, D(),G$
110 CHAIN "PROG3", 10
```

line 100 passes variables A, B, C, D and G$ to the CHAIN command in line 110.

See also CHAIN.

CONT

CONT	**Statement**

Resumes program execution.

You may only use CONT if the program was stopped by the (BREAK) key, a STOP or an END statement in the program.

CONT is primarily a debugging tool. During a break or stop in execution, you may examine variable values (using PRINT) or change these values. Then type CONT (ENTER); execution continues with the current variable values.

You cannot use CONT after editing your program lines or otherwise changing your program. CONT is also invalid after execution has ended normally.

Example

```
10 INPUT A, B, C
20 K=A^2
30 L=B^3/ .26
40 STOP
50 M=C+40*K+100: PRINT M
```

Run this program. (To enter the ^, press (CLEAR) (;).) You will be prompted with:

```
?
```

Type:

```
1, 2, 3 (ENTER)
```

The computer displays:

```
Break in 40
```

You can now type any immediate command.

For example:

```
PRINT L
```

displays 30.7692. You can also change the value of A, B, or C.

For example:

```
C = 4
```

changes the value of C in the program. Type:

```
CONT
```

your screen displays: 144.

See also STOP.

COS

COS(*number*)	**Function**

Computes the cosine of *number*.

COS returns the cosine of *number* in radians. The *number* must be given in radians. When *number* is in degrees, use COS(*number* * .01745329).

The result is always single precision.

Examples

```
Y = COS(X * .01745329)
```

stores in Y the cosine of X, if X is an angle in degrees.

```
PRINT COS(5.8) - COS(85 * .42)
```

prints the arithmetic (not trigonometric) difference of the two cosines.

CSNG

CSNG(*number*)	**Function**

Converts *number* to single precision.

If *number* is double precision, when its single-precision value is printed, only six significant digits are shown. BASIC rounds the number in this conversion.

Example

```
PRINT CSNG(.1453885509)
```

prints .145389

Sample Program

```
280 V# = 876.2345678#
290 PRINT V#; CSNG(V#)
RUN
 876.2345678000001     876.235
Ready
```

CVD, CVI, CVS

CVD(*eight-byte string*) **CVS(*four-byte string*)** **CVI(*two-byte string*)**	**Function**

Convert string values to numeric values.

These functions let you restore data to numeric form after it is read from disk. Typically, the data has been read by a GET statement, and is stored in a direct access file buffer.

CVD converts an *eight-byte string* to a double-precision number. CVS converts a *four-byte string* to a single-precision number. CVI converts a *two-byte string* to an integer.

CVD, CVI, and CVS are the inverses of MKD$, MKI$, and MKS$, respectively.

Examples

Suppose the name GROSSPAY$ references an eight-byte field in a direct-access file buffer, and after GETting a record, GROSSPAY$ contains an MKD$ representation of the number 13123.38. Then the statement

```
A# = CVD(GROSSPAY$)
```

assigns the numeric value 13123.38 to the double-precision variable A#.

Sample Program

This program reads from the file "TEST/DAT", which is assumed to have been previously created. For the program that creates the file, see MKD$, MKI$, and MKS$.

```
1420 OPEN "D", 1, "TEST/DAT", 14
1430 FIELD 1, 2 AS I1$, 4 AS I2$, 8 AS I3$
1440 GET 1
1450 PRINT CVI(I1$), CVS(I2$), CVD(I3$)
1460 CLOSE
```

NOTE: GET without a record number tells BASIC to get the first record from the file, or the record following the last record accessed.

DATA

DATA *constant, . . .*	**Statement**

Stores numeric and string *constants* to be accessed by a READ statement.

This statement may contain as many *constants* (separated by commas) as will fit on a line. Each will be read sequentially, starting with the first constant in the first DATA statement, and ending with the last item in the last DATA statement.

Numeric expressions are not allowed in a DATA list. If your string values include leading blanks, colons, or commas, you must enclose these values in double quotation marks.

DATA statements may appear anywhere it is convenient in a program. The data types in a DATA statement must match up with the variable types in the corresponding READ statement, otherwise a "Syntax error" occurs.

Examples

```
1340 DATA NEW YORK, CHICAGO, LOS ANGELES,
     PHILADELPHIA, DETROIT
```

stores five string data items. Note that quote marks aren't needed, since the strings contain no delimiters and the leading blanks are not significant.

```
1350 DATA 2.72, 3.14159, 0.0174533, 57.29578
```

stores four numeric data items.

```
1360 DATA "SMITH, T.H.", 38, "THORN, J.R.", 41
```

stores both types of constants. Quote marks are required around the first and third items because they contain commas (commas are delimiters within data fields).

Sample Program

```
NEW
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
```

This program READS string and numeric data from the DATA statement in line 30.

DATE$

DATE$	**Function**

Returns today's date.

The operator sets the date when TRSDOS is started up.
(This system supports dates between January 1, 1980 and December 3, 1987).

During a program, if you request the date, BASIC displays it in this fashion:

```
03/12/83
```

Sample Program

```
1090 PRINT "Inventory Check:"
1100 IF DATE$ = "01/31/80" THEN PRINT "Today is
     the last day of January 1980. Time to
     perform monthly inventory.": END
```

DEFDBL/INT/SNG/STR

Statement

DEFDBL ***letter, . . .***
DEFINT ***letter, . . .***
DEFSNG ***letter, . . .***
DEFSTR ***letter, . . .***

Defines any variables beginning with *letter(s)* as: (DBL) double precision, (INT) integer, (SNG) single precision, or (STR) string.

NOTE: A type declaration character always takes precedence over a DEF statement.

Examples

```
10 DEFDBL L-P
```

classifies all variables beginning with the letters L through P as double-precision variables. Their values include 17 digits of precision, though only 16 are printed out.

```
10 DEFSTR A
```

classifies all variables beginning with the letter A as string variables.

```
10 DEFINT I-N, W,Z
```

classifies all variables beginning with the letters I through N, W and Z as integer variables. Their values are in the range −32768 to 32767.

```
10 DEFSNG I, Q-T
```

classifies all variables beginning with the letters I or Q through T as single-precision variables. Their values include seven digits of precision, though only six are printed out.

DEF FN

	Statement
DEF FN *function name* [(*variable*, . . .)] =*function definition*	

Defines *function name* according to your *function definition*.

Function name must be a valid variable name. The type of variable used determines the type of value the function will return. For example, if you use a single-precision variable, the function will always return single-precision values.

Variable represents those variables in *function definition* that are to be replaced when the function is called. If you enter several variables, separate them by commas.

Function definition is an expression that performs the operation of the function. A variable used in a function definition may or may not appear as *variable*. If it does, BASIC uses its value to perform the function. Otherwise, it uses the current value of the variable.

Once you define and name a function (by using this statement), you can call it and BASIC performs the associated operations.

Examples

```
DEF FNR = RND(90)+9
```

defines a function FNR to return a random value between 10 and 99. Notice that the function can be defined with no arguments.

```
210 DEF FNW# (A#,B#)=(A#-B#)*(A#-B#)
280 T = FNW#(I#,J#)
```

defines function FNW# in line 210. Line 280 calls that function and replaces parameters A# and B# with parameters I# and J#. (We assume that I# and J# were assigned values elsewhere in the program.)

NOTE: Using a variable as a parameter in a DEF FN statement has no effect on the value of that variable. You may use that variable in another part of the program without interference from DEF FN.

DEF USR

DEF USR[*digit*]=*address*	**Statement**

Defines the starting address for the assembly-language subroutine identified by *digit*.

A program may contain any number of DEF USR statements, allowing access to as many subroutines as necessary. However, only 10 definitions may be in effect at one time.

If you omit *digit*, BASIC assumes USR0.

See also USR, VARPTR and CALL.

Examples

```
DEF USR3 = &HF700
```

assigns the starting address F700 hexadecimal, 63231 decimal, to the USR3 call. When your program calls USR3, control branches to your subroutine beginning at F700.

```
DEF USR = (BASE + 16)
```

assigns the starting address of BASE + 16 to the USR0 subroutine.

DELETE

DELETE *line1 - line2*	**Statement**

Deletes from *line1* through *line2* of a program in memory.

A period (".") can be substituted for either *line1* or *line2* to indicate the current line number.

Examples

```
DELETE 70
```

deletes line 70 from memory. If there is no line 70, an error will occur.

```
DELETE 50-110
```

deletes lines 50 through 110 inclusive.

```
DELETE -40
```

deletes all program lines up to and including line 40.

```
DELETE -.
```

deletes all program lines up to and including the line that has just been entered or edited.

```
DELETE .
```

deletes the program line that has just been entered or edited.

DIM

DIM *array* (*dimension(s)*), *array* (*dimension(s)*), . . .	**Statement**

Sets aside storage for *arrays* with the *dimensions* you specify.

Arrays may be of any type: string, integer, single precision or double precision, depending on the type of variable used to name the array. If no type is specified, the array is classified as single precision.

When you create the array, BASIC reserves space in memory for each element of the array. All elements in a newly-created array are set to zero (numeric arrays) or the null string (string arrays).

NOTE: The lowest element in a dimension is always zero, unless OPTION BASE 1 has been used.

Arrays can be created implicitly, without explicit DIM statements. Simply refer to the desired array in a BASIC statement. For example,

```
A(5) = 300
```

creates array A and assigns element A(5) the value of 300. Each dimension of an implicitly-defined array is 11 elements deep, subscripts 0 – 10.

Examples

```
DIM AR(100)
```

sets up a one-dimensional array AR( ), containing 101 elements: AR(0), AR(1), AR(2), . . . , AR(98), AR(99), and AR(100).

NOTE: The array AR( ) is completely independent of the variables AR.

```
DIM L1%(8,25)
```

sets up a two-dimensional array L1%(,), containing 9 × 26 integer elements, L1%(0,0), L1%(1,0), L1%(2,0), . . . ,L1%(8,0), L1%(0,1), L1%(1,1), . . . ,L1%(8,1), . . . ,L1%(0,25), L1%(1,25), . . . , L1%(8,25).

Two-dimensional arrays like AR(,) can be thought of as a table in which the first subscript specifies a row position, and the second subscript specifies a column position:

```
Ø,Ø   Ø,1   Ø,2   Ø,3   ...   Ø,23   Ø,24   Ø,25
1,Ø   1,1   1,2   1,3   ...   1,23   1,24   1,25
.
.
.
7,Ø   7,1   7,2   7,3   ...   7,23   7,24   7,25
8,Ø   8,1   8,2   8,3   ...   8,23   8,24   8,25
```

```
DIM B1(2,5,8), CR(2,5,8), LY$(5Ø,2)
```

sets up three arrays:

B1(,,) and CR (, ,) are three-dimensional, each containing 3*6*9 elements.

LY(,) is two-dimensional, containing 51*3 string elements.

EDIT

EDIT *line*	**Statement**

Enters the edit mode so that you can edit the statement on *line*.

See the chapter on the "Edit Mode" for more information.

Examples

```
EDIT 1ØØ
```

enters edit mode at line 1ØØ.

```
EDIT .
```

enters edit mode at current line.

END

END	Statement

Ends execution of a program.

This statement may be placed anywhere in the program. It forces execution to end at some point other than the last sequential line.

An END statement at the end of a program is optional.

Sample Program

```
40 INPUT S1, S2
50 GOSUB 100
55 PRINT H
60 END
100 H=SQR(S1*S1 + S2*S2)
110 RETURN
```

line 60 prevents program control from "crashing" into the subroutine. Line 100 may only be accessed by a branching statement, such as GOSUB in line 50.

EOF

EOF(*buffer*)	Function

Detects the end of a file.

This function checks to see whether all characters up to the end-of-file marker have been accessed, so you can avoid "Input past end" errors during sequential input.

EOF(*buffer*) returns 0 (false) when the EOF record has not been read yet, and −1 (true) when it has been read. The buffer number must access an open file.

Sample Program

The following sequence of lines reads numeric data from DATA/TXT into the array A( ). When the last data character in the file is read, the EOF test in line 30 "passes", so the program branches out of the disk access loop.

```
1470 DIM A(100)     'ASSUMING THIS IS A SAFE VALUE
1480 OPEN "I", 1, "DATA/TXT"
1490 I% = 0
1500 IF EOF(1) THEN 1540
1510 INPUT #1, A(I%)
1520 I% = I% + 1
1530 GOTO 1500
1540 REM   PROG.  CONT.  HERE AFTER DISK INPUT
```

ERASE

ERASE ***array, . . .*** **Statement**

Erases one or more *arrays* from a program.

This lets you to either redimension arrays or use their previously allocated space in memory for other purposes.

If one of the parameters of ERASE is a variable name which is not used in the program, an "Illegal Function Call" occurs.

Example

```
450 ERASE C,F
460 DIM F(99)
```

line 450 erases arrays C and F. Line 460 redimensions array F.

ERL

ERL	Statement

Returns the line in which an error has occurred.

This function is primarily used inside an error-handling routine. If no error has occurred when ERL is called, line number 0 is returned. Otherwise, ERL returns the line number in which the error occurred. If the error occurred in the command mode, 65535 (the largest number representable in two bytes) is returned.

Examples

```
PRINT ERL
```

prints the line number of the error.

```
E = ERL
```

stores the error's line number for future use.

For an example of how to use ERL in a program, see ERROR.

ERR

ERR	Statement

Returns the error code (if an error has occurred).

ERR is only meaningful inside an error-handling routine accessed by ON ERROR GOTO. See Appendix D for a list of Error Codes.

Example

```
IF ERR = 7 THEN 1000 ELSE 2000
```

branches the program to line 1000 if the error is an "Out of Memory" error (code 7); if it is any other error, control goes instead to line 2000.

For an example of how to use ERR in a program, see ERROR.

ERRS$

ERRS$	Function

Returns a system error number and message.

This function returns the number and description of the TRSDOS error that caused the latest BASIC disk-related error. If no TRSDOS error has occurred, ERRS$ returns a null string.

Example

```
PRINT "THE LATEST TRSDOS ERROR IS "; ERRS$
```

prints the latest error number message.

ERROR

ERROR *code*	**Statement**

Simulates a specified error during program execution.

Code is an integer expression in the range 0 to 255 specifying one of BASIC's error codes.

This statement is mainly used for testing an ON ERROR GOTO routine. When the computer encounters an ERROR code statement, it proceeds as if the error corresponding to that code had occurred. (Refer to Appendix D for a listing of Error Codes and their meanings).

Example

```
ERROR 1
```

a "Next Without For" error (code 1) "occurs" when BASIC reaches this line.

Sample Program

```
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET"; B
130 IF  B>5000 THEN ERROR 21 ELSE GOTO 420
400 IF ERR = 21 THEN PRINT "HOUSE LIMIT IS  $5000"
410 IF ERL = 130 THEN RESUME 500
420 S = S+B
430 GOTO 120
500 PRINT "THE TOTAL AMOUNT OF YOUR BET IS";S
510 END
```

This program receives and totals bets until one of them exceeds the house limit.

EXP

EXP(*number*) **Function**

Calculates the natural exponent of *number*.

Returns e (base of natural logarithms) to the power of *number*. This is the inverse of the LOG function; therefore, *number* = EXP(LOG(*number*)). The *number* you supply must be less than or equal to 87.3365.

The result is always single precision.

Example

```
PRINT EXP(-2)
```

prints the exponential value .135335.

Sample Program

```
310 INPUT "NUMBER"; N
320 PRINT "E RAISED TO THE N POWER IS" EXP(N)
```

FIELD

FIELD ***buffer, length*** **AS** ***field name, . . .***	**Statement**

Divides a direct-access *buffer* into one or more fields. Each field is identified by *field name* and is the *length* you specify.

Field name must be a string variable.

This divides a direct file buffer so that you can send data from memory to disk and disk to memory. FIELD must be run prior to GET or PUT.

Before "fielding" a buffer, use an OPEN statement to assign that buffer to a particular disk file. (The direct access mode, i.e., OPEN "D", . . . must be used.) The sum of all field lengths should equal the record length assigned when the file was OPENed.

You may use the FIELD statement any number of times to "re-field" a file buffer. "Fielding" a buffer does not clear the buffer's contents; only the means of accessing it. Also, two or more field names can reference the same area of the buffer.

See also the chapter on "Disk Files", OPEN, CLOSE, PUT, GET, LSET, and RSET.

Example

```
FIELD 3, 128  AS A$, 128 AS B$
```

tells BASIC to assign two 128-byte fields to the variables A$ and B$. If you now print A$ or B$, you will see the contents of the field. Of course, this value would be meaningless unless you have previously used GET to read a 256-byte record from disk.

NOTE: All data — both strings and numbers — must be placed into the buffer in string form. There are three pairs of functions (MKI$/CVI, MKS$/CVS, and MKD$/CVD) for converting numbers to strings and strings to numbers.

```
FIELD 3, 16 AS NM$, 25 AS AD$, 10 AS CY$, 2 AS
ST$, 7 AS ZP$
```

assigns the first 16 bytes of buffer 3 to field NM$; the next 25 bytes to AD$; the next 10 to CY$; the next 2 to ST$; and the next 7 to ZP$.

FIX

FIX(*number*)	**Function**

Returns the truncated integer of *number*.

All digits to the right of the decimal point are simply chopped off, so the resultant value is a whole number. For a negative, non-whole number X, FIX(X) = INT(X) + 1. For all others, FIX(X) = INT(X).

The result is the same precision as the argument (except for the fractional portion).

Examples

```
PRINT FIX (2.6)
```

prints 2.

```
PRINT FIX(-2.6)
```

prints -2.

FOR/NEXT

Statement

FOR *variable* = *initial value* TO *final value* [**STEP *increment***]
NEXT [***variable***]

Establishes a program loop.

A loop allows for a series of program statements to be executed over and over a specified number of times.

BASIC executes the program lines following the FOR statement until it encounters a NEXT. At this point, it increases *variable* by STEP *increment*. If the value of *variable* is less than or equal to *final value*, BASIC branches back to the line after FOR, and repeats the process. If *variable* is greater than *final value*, it completes the loop and continues with the statement after NEXT.

If *increment* has a negative value, then the final value of *variable* is actually lower than the initial value. BASIC always sets the final value for the loop variable before setting the initial value.

NOTE: BASIC skips the body of the loop if *initial value* times the sign of STEP *increment* exceeds *final value* times the sign of STEP *increment*.

Example

```
20 FOR H=1 TO -10 STEP -2
30 PRINT H
40 NEXT H
```

the initial value of H times the sign of STEP increment is greater than the final value of H times the sign of STEP increment, therefore BASIC skips the body of the loop. (The sign of STEP increment is negative in this case.)

Sample Program

```
820 I=5
830 FOR I = 1 TO I + 5
840 PRINT I;
850 NEXT
RUN
```

this loop is executed ten times. It produces the following output:

```
1  2  3  4  5  6  7  8  9  10
```

Nested Loops

FOR/NEXT loops may be "nested". That is, a FOR . . . NEXT loop may be placed within the context of another FOR . . . NEXT loop.

The NEXT statement for the inside loop must appear before the NEXT for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

Sample Program

```
880 FOR I = 1 TO 3
890 PRINT "OUTER LOOP"
900 FOR J = 1 TO 2
910 PRINT "INNER LOOP"
920 NEXT J
930 NEXT I
```

This program performs three "outer loops" and within each, two "inner loops".

The NEXT statement can be used to close nested loops by listing the counter variables (but make sure not to type the variables out of order). For example, delete line 920 and change 930 to:

```
NEXT J, I
```

NOTE: In nested loops, if the variable(s) in the NEXT statement is omitted, the NEXT statement matches the most recent FOR statement.

FRE

FRE(*dummy number*) or (*dummy string*) **Function**

Returns the number of bytes in memory not being used by BASIC.

Because of the ways BASIC handles memory, the best way to ensure accurate results from FRE and MEM is to use them at the beginning of your program before it performs any string assignments.

The values returned by FRE and MEM can be deceiving after string assignments. For example, type these lines:

```
NEW (ENTER)
PRINT FRE(0) (ENTER)
```

Notice the number of bytes returned by FRE. Now, assign a string to A$, by typing:

```
A$="THIS IS A STRING THAT WILL REQUIRE MEMORY"
(ENTER)
PRINT FRE (0) (ENTER)
```

You can see that some of the bytes are no longer free. They were used to create A$ and to store its contents.

Now, assign fewer bytes to A$, by typing:

```
A$="SHORT STRING" (ENTER)
```

Check the free space again:

```
PRINT FRE(0) (ENTER)
```

Even though fewer bytes are now assigned to A$, the amount of free memory does **not** increase; it decreases. This is because BASIC discards the section of memory that contains the original string and allocates a new section to contain the new string.

If BASIC continues to perform string assignments indefinitely, it eventually runs out of free space. If you use the FRE function, it indicates this. But, in fact, there **is** memory left—namely, those sections discarded earlier, which contain strings no longer assigned to variables. When you try to assign another string, BASIC scans memory, finds these sections, and adds all of them back into free memory. This is called *garbage collection*. It occurs automatically whenever you execute RUN, NEW, LOAD, or CLEAR, as well as whenever all memory appears to be used.

If you use FRE immediately after a garbage collection, it again returns the true amount of memory left.

NOTE: The error message "Out of string space" appears if BASIC is unable to find enough room for a string, even after a garbage collection.

Examples

```
PRINT FRE("44")
```

prints the amount of memory left.

```
PRINT FRE(44)
```

prints the amount of memory left.

GET

GET ***buffer*** [***,record***]	**Statement**

Gets a *record* from a direct-access disk file and places it in a *buffer*.

Before using GET, you must OPEN the file and assign it a buffer.

When BASIC encounters GET, it reads the record number from the file and places it into the buffer. The actual number of bytes read equals the record length set when the file is OPENed.

If *record* is omitted, BASIC gets the next record (after the last GET) and reads it into the buffer.

Examples

```
GET 1
```

gets the next record into buffer 1.

```
GET 1, 25
```

gets record 25 into buffer 1.

GOSUB

GOSUB *line*	**Statement**

Goes to a subroutine, beginning at *line*.

You can call subroutine as many times as you want. When the computer encounters RETURN in the subroutine, it returns control to the statement which follows GOSUB.

GOSUB is similar to GOTO in that it may be preceded by a test statement. Every subroutine must end with a RETURN.

Example

```
GOSUB 1000
```

branches control to the subroutine at 1000.

Sample Program

```
260 GOSUB 280
270 PRINT "BACK FROM SUBROUTINE": END
280 PRINT "EXECUTING THE SUBROUTINE"
290 RETURN
```

transfers control from line 260 to the subroutine beginning at line 280. Line 290 instructs the computer to return to the statement immediately following GOSUB.

GOTO

GOTO *line*	Statement

Goes to the specified *line*.

When used alone, GOTO *line* results in an unconditional (automatic) branch. However, test statements may precede the GOTO to effect a conditional branch.

You can use GOTO in the command mode as an alternative to RUN. This lets you pass values assigned in the command mode to variables in the execute mode.

Example

```
GOTO 100
```

transfers control automatically to line 100.

Sample Program

```
10 READ R
20 IF R = 13 THEN END
30 PRINT "R=";R
40 A=3.14*R^2
50 PRINT "AREA =";A
60 GOTO 10
70 DATA 5,7,12, 13
RUN
```

Line 10 reads each of the data items in line 60; line 50 returns program control to line 10. This enables BASIC to calculate the area for each of the data items, until it reaches item 13.

NOTE: To enter the ^ symbol, press CLEAR ;.

HEX$

HEX$(*number*)	Function

Calculates the hexadecimal value of *number*.

HEX$ returns a string which represents the hexadecimal value of the argument. The value returned is like any other string: it cannot be used in a numeric expression. That is, you cannot add hex strings. You can concatenate them, though.

Examples

```
PRINT HEX$(30), HEX$(50), HEX$(90)
```

prints the following strings:

```
1E          32          5A
```

```
Y$ = HEX$(X/16)
```

Y$ is the hexadecimal string representing the integer quotient X/16.

IF . . . THEN . . . ELSE

IF *expression* THEN *statement(s)* or *line* **[ELSE *statement(s)* or *line*]**	**Statement**

Tests a conditional expression and makes a decision regarding program flow.

If *expression* is true, control proceeds to the THEN *statement* or *line*. If not, control jumps to the matching ELSE *statement*, *line*, or down to the next program line.

Examples

```
IF X > 127 THEN PRINT "OUT OF RANGE" : END
```

passes control to PRINT, then to END if X is greater than 127. If X is not greater than 127, control jumps down to the next line in the program, skipping the PRINT and END statements.

```
If A < B THEN PRINT "A < B" ELSE PRINT "B <= A"
```

tests the first expression, if true, prints "A < B". Otherwise, the program jumps to the ELSE statement and prints "B <= A".

```
IF X > 0 AND Y <> 0 THEN Y = X + 180
```

assigns the value X + 180 to Y if both expressions are true. Otherwise, control passes directly to the next program line, skipping the THEN clause.

```
IF A$ = "YES" THEN 210 ELSE IF A$ = "NO" THEN 400
    ELSE 370
```

branches to line 210 if A$ is YES. If not, the program skips over to the first ELSE, which introduces a new test. If A$ is NO, then the program branches to line 400. If A$ is any value besides NO or YES, the program branches to line 370.

Sample Program

IF THEN ELSE statements may be nested. However, you must take care to match up the IFs and ELSEs. (If the statement does not contain the same number of ELSE's and IF's, each ELSE is matched with the closest unmatched IF.)

```
1040 INPUT "ENTER TWO NUMBERS"; A, B
1050 IF A <= B THEN IF A < B THEN PRINT A;
     ELSE PRINT "NEITHER"; ELSE PRINT B;
1060 PRINT "IS SMALLER THAN THE OTHER"
```

This program prints the relationship between the two numbers entered.

INKEY$

INKEY$ **Function**

Returns a keyboard character.

Returns a one-character string from the keyboard without having to press ENTER. If no key is pressed, a null string (length zero) is returned. Characters typed to INKEY$ are not echoed to the display.

INKEY$ is invariably put inside some sort of loop. Otherwise a program execution would pass through the line containing INKEY$ before a key could be pressed.

Example

```
10 A$ = INKEY$
20 IF A$ = "" THEN 10
```

This causes the program to wait for a key to be pressed.

INP

INP(*port*) **Function**

Returns the byte read from a *port*.

INP is the complementary function of the OUT statement.

Port may be any integer from 0 to 255. For information on assigned ports, see the *Model 4/4P Technical Reference Manual*.

Example

```
100 A=INP(42)
```

INPUT

INPUT [*prompt string;*] **variable1, variable2, . . .**	**Statement**

Inputs data from the keyboard into one or more *variables*.

When BASIC encounters this statement, it stops execution and displays a question mark. This means that the program is waiting for you to type data.

INPUT may specify a list of string or numeric variables, indicating string or numeric data items to be input. For instance, INPUT X$, X1, Z$, Z1 calls for you to input a string literal, a number, another string literal, and another number, in that order.

The number of data items you supply must be the same as the number of variables specified. You must separate data items by commas.

Responding to INPUT with too many items, or with the wrong type of value (including numeric type), causes BASIC to print the message "?Redo from start". No values are assigned until you provide an acceptable response.

If a *prompt string* is included, BASIC prints it, followed by a question mark. This helps the person inputting the data to enter it correctly. If instead of a semicolon, you type a comma after *prompt string*, BASIC suppresses the question mark when printing the prompt. *Prompt string* must be enclosed in quotes. It must be typed immediately after INPUT.

Examples

```
INPUT Y%
```

when BASIC reaches this line, you must type any number and press (ENTER) before the program will continue.

```
INPUT SENTENCE$
```

when BASIC reaches this line, you must type in a string. The string wouldn't have to be enclosed in quotation marks unless it contained a comma, a colon, or a leading blank.

```
INPUT "ENTER YOUR NAME AND AGE (NAME, AGE)";
N$, A
```

would print a message on the screen which would help the person at the keyboard to enter the right kind of data.

Sample Program

```
50 INPUT "HOW MUCH DO YOU WEIGH"; X
60 PRINT "ON MARS YOU WOULD WEIGH ABOUT"
   CINT(X * .38) "POUNDS."
```

INPUT#

INPUT ***#buffer, variable, . . .***	**Statement**

Inputs data from a sequential disk file and stores it in a program *variable*.

Buffer is the number used when the file was OPENed for input.

Variable contains the variable name(s) that will be assigned to the item(s) in the file.

With INPUT#, data is input sequentially. That is, when the file is OPENed, a pointer is set to the beginning of the file. The pointer advances each time data is input. To start reading from the beginning of the file again, you must close the file buffer and re-OPEN it.

INPUT# doesn't care how the data was placed on the disk — whether a single PRINT# statement put it there, or whether it required ten different PRINT# statements. What matters to INPUT# is the position of the terminating characters and the EOF marker.

When inputting data into a variable, BASIC ignores leading blanks. When the first non-blank character is encountered, BASIC assumes it has encountered the beginning of the data item.

The data item ends when a terminating character is encountered or when a terminating condition occurs. The terminating characters vary, depending on whether BASIC is inputting to a numeric or string variable.

Numeric values: BASIC begins input at the first character which is neither a space nor a carriage return. It ends input when it encounters a space, carriage return, or a comma.

String values: BASIC begins input with the first character which is neither a space nor carriage return. It ends input when it encounters a carriage return or comma. One exception to this rule: If the first character is a quotation mark("), the string will consist of all characters between the first quotation mark and the second. Thus, a quoted string may not contain a quotaton mark as a character.

If the end-of-file is reached when a numeric or string item is being INPUT, the item is terminated.

Examples:

```
INPUT #1,A,B
```

sequentially inputs two numeric data items from disk and places them in A and B. Buffer #1 is used.

```
INPUT #4,A$,B$,C$
```

sequentially inputs three string data items from disk and places them in A$, B$, and C$. Buffer #4 is used.

INPUT$

INPUT$(*number* [,*buffer*])	**Statement**

Inputs a string of characters from either the keyboard or a sequential disk file.

Number is the number of characters to be input. It must be a value in the range 1 to 255. *Buffer* is a buffer which accesses a sequential input file.

INPUT$(*number*) inputs a string of characters from the keyboard. When the program reaches this line, it stops until you (or any operator) type *number* characters. (You don't need to press (ENTER) to signify end-of-line.) The character(s) you type are not displayed on the screen. Any character, except (BREAK), is accepted for input. No characters are echoed.

INPUT$(*number*, *buffer*) inputs a string from a sequential disk file. *Buffer* is the buffer associated with that disk file.

Examples

```
A$ = INPUT$(5)
```

assigns a string of five keyboard characters to A$. Program execution is halted until the operator types five characters.

```
A$ = INPUT$(11,3)
```

assigns a string of 11 characters to A$. The characters are read from the disk file associated with buffer 3.

Sample Programs

This program shows how you could use INPUT$ to have an operator input a password for accessing a protected file. By using INPUT$, the operator can type in the password without anyone seeing it on the video display. (To see the full file specification, run the program, then type PRINT F$.)

```
110 LINE INPUT "TYPE IN THE FILESPEC/EXT"; F$
120 PRINT "TYPE IN THE PASSWORD -- MUST TYPE
    8 CHARACTERS: ";
130 P$ = INPUT$(8)
140 F$ = F$ * "." * P$
```

In the program below, line 100 OPENs a sequential input file (which we assume has been previously created). Line 200 retrieves a string of 70 characters from the file and stores them in T$. Line 300 CLOSEs the file.

```
100 OPEN "I", 2, "TEST/DAT"
200 T$ = INPUT$(70,2)
300 CLOSE
```

INSTR

INSTR([*integer*,] *string1*, *string2*)	**Function**

Searches for the first occurrence of *string2* in *string1*, and returns the position at which the match is found.

Integer specifies a position in *string1*. If used it must be a value in the range 1 to 255.

This function lets you search through a string to see if it contains another string. If it does, INSTR returns the starting position of the substring in the target string; otherwise, it returns zero. Note that the entire substring must be contained in the search string, or zero is returned.

Optional *integer* sets the position for starting the search. If omitted, INSTR starts searching at the first character in *string1*.

Examples

In these examples, A$ = "LINCOLN":

```
INSTR(A$, "INC")
```

returns a value of 2.

```
INSTR(A$, "12")
```

returns a zero.

```
INSTR(A$, "LINCOLNABRAHAM")
```

returns a zero. For a slightly different use of INSTR, look at:

```
INSTR(3, "1232123", "12")
```

which returns 5.

Sample Program

The program below uses INSTR to search through the addresses contained in the program's DATA lines. It counts the number of addresses with a specified county zip code (761—) and returns that number. The zip code is preceded by an asterisk to distinguish it from the other numeric data found in the address.

```
360 RESTORE
370 COUNTER = 0
390 READ ADDRESS$
395 IF ADDRESS$ = "$END" THEN 410
400 IF INSTR(ADDRESS$, "*761") <> 0 THEN COUNTER =
    COUNTER + 1 ELSE 390
405 GOTO 390
410 PRINT "NUMBER OF TARRANT COUNTY, TX
    ADDRESSES IS" COUNTER: END
420 DATA "5950 GORHAM DRIVE, BURLESON, TX  *76148"
430 DATA "71 FIRSTFIELD ROAD, GAITHERSBURG, MD
    *20760"
440 DATA "1000 TWO TANDY CENTER, FORT WORTH,
    TX *76102"
450 DATA "16633 SOUTH CENTRAL EXPRESSWAY,
    RICHARDSON, TX *75080"
460 DATA "$END"
```

INT

INT(*number*)	**Function**

Converts *number* to integer value.

This function returns the largest integer which is not greater than the *number*. *Number* may be an expression.

The result has the same precision as the argument except for the fractional portion. *Number* is not limited to the range −32768 to 32767.

Examples

```
PRINT INT(79.89)
```

prints 79.

```
PRINT INT(-12.11)
```

prints −13.

KILL

KILL "*filespec*"	**Statement**

"Kills" (deletes) *filespec* from disk.

You may KILL any type of disk file. However, if the file is currently OPEN, a "File already open" error occurs. You must CLOSE the file before deleting it.

Example

```
KILL "FILE/BAS"
```

deletes this file from the first drive which contains it.

```
KILL "DATA:2"
```

deletes this file from Drive 2 only. BASIC does not check the other drives.

LEFT$

LEFT$(*string,integer*)	**Function**

Returns the leftmost *integer* characters of *string*.

If *integer* is equal to or greater than LEN (string), the entire string is returned.

Examples:

```
PRINT LEFT$("BATTLESHIPS", 6)
```

prints BATTLE.

```
PRINT LEFT$("BIG FIERCE DOG", 20)
```

since BIG FIERCE DOG is less than 20 characters long, the whole phrase is printed.

Sample Program

```
740 A$ = "TIMOTHY"
750 B$ = LEFT$(A$, 3)
760 PRINT B$; "--THAT'S SHORT FOR "; A$
```

When this is run, BASIC prints:

```
TIM--THAT'S SHORT FOR TIMOTHY
```

Line 750 gets the three leftmost characters of A$ and stores them in B$. Line 760 prints these three characters, a string, and the original contents of A$.

LEN

LEN(*string*)	**Function**

Returns the number of characters in *string*.

Examples

```
X = LEN(SENTENCE$)
```

gets the length of SENTENCE$ and stores it in X.

```
PRINT LEN("CAMBRIDGE") + LEN("BERKELEY")
```

prints 17.

LET

[LET] *variable* = *expression*	**Statement**

Assigns the value of *expression* to *variable.*

BASIC doesn't require assignment statements to begin with LET, but you might want to use LET to be compatible with versions of BASIC that do require it.

Examples

```
LET A$ = "A ROSE IS A ROSE"
LET B1 = 1.23
LET X  = X - Z1
```

In each case, the variable on the left side of the equals sign is assigned the value of the constant or expression on the right side.

Sample Program

```
550 P = 1001: PRINT "P =" P
560 LET P = 2001: PRINT "NOW P = "P
```

LINE INPUT

LINE INPUT[*prompt string*;] ***string variable***	**Statement**

Inputs an entire line (up to 254 characters) from the keyboard.

LINE INPUT is a convenient way to input string data without having to worry about accidental entry of delimiters (commas, quotation marks, etc.).

LINE INPUT (the space is *not* optional) is similar to INPUT, except:

— The computer does not display a question mark when waiting for input.
— Each LINE INPUT statement can assign a value to only one variable.
— Commas and quotes can be used as part of the string input.
— Leading blanks are not ignored — they become part of variable.

The only way to terminate the string input is to press (ENTER).

Some situations require that you input commas, quotes, and leading blanks as part of the data. LINE INPUT serves well in such cases.

Examples:

```
LINE INPUT A$
```

inputs A$ without displaying any prompt.

```
LINE INPUT "LAST NAME, FIRST NAME? "; N$
```

displays a prompt message and inputs data. Commas do not terminate the input string, as they would in an INPUT statement.

You may abort a LINE INPUT statement by pressing (BREAK). BASIC returns to command level and displays Ready. Typing CONT resumes execution at LINE INPUT.

LINE INPUT#

LINE INPUT *#buffer, variable*	**Statement**

Inputs an entire line of data from a sequential disk file to a string *variable*.

Buffer is the number under which the file was OPENed.

This statement is useful when you want to read an ASCII-format BASIC program file as data, or when you want to read in data without following the usual restrictions regarding leading characters and terminators.

LINE INPUT# reads everything from the first character up to:

— the end-of-file
— the 255th data character

Other characters encountered — quotes, commas, leading blanks — are included in the string.

Example

If the data on disk looks like this:

```
10 CLEAR 500
20 OPEN "I", 1, "PROG"
```

then the statement

```
LINE INPUT #1, A$
```

could be used repetitively to read each program line, one at a time.

LIST

LIST [*startline*]-[*endline*]	**Statement**

Lists a program in memory to the display.

Startline specifies the first line to be listed. If omitted, BASIC starts with the first line in your program.

Endline specifies the last line to be listed. If omitted, BASIC ends with the last line in your program.

You can substitute period (.) for either *startline* or *endline* to signify current line number.

Examples

```
LIST
```

displays the entire program. To stop the automatic scrolling, press SHIFT @. This freezes the display. Press any key to continue the listing.

```
LIST 50
```

displays line 50.

```
LIST 50-85
```

displays lines in the range 50-85.

```
LIST .-
```

displays the program line that has just been entered or edited, and all higher-numbered lines.

```
LIST -227
```

displays all lines up to and including 227.

LLIST

LLIST [*startline*]-[*endline*]	**Statement**

Lists program lines in memory to the printer.

The only difference between LLIST and LIST is that LLIST lists the lines on printer. See LIST.

Examples

```
LLIST
```

lists the entire program to the printer. To stop this process, press (SHIFT)(@). This causes a temporary halt in the computer's output to the printer. Press any key to continue printing.

```
LLIST 68-90
```

prints lines in the range 68-90.

LOAD

LOAD ***"filespec"*** [**,R**]	**Statement**

Loads *filespec*, a BASIC program, into memory.

The R option tells BASIC to run the program. (LOAD with the R option is equivalent to the command RUN *filespec*, R.)

LOAD without the R option wipes out any resident BASIC program, clears all variables, and CLOSES all OPEN files. LOAD with the R option leaves all OPEN files open and runs the program automatically.

You can use either of these commands inside programs to allow program chaining (one program calling another).

If you attempt to LOAD a non-BASIC file, a "Direct statement in file" error will occur.

Example

```
LOAD "PROG1/BAS:2"
```

loads PROG1/BAS from Drive 2. BASIC then returns to the command mode.

```
LOAD "PROG1/BAS"
```

loads PROG1/BAS. Since no drive is specified, BASIC begins searching for it in Drive 0.

LOC

LOC(*buffer*)	**Function**

Returns the current record number.

Buffer is the buffer under which the file was OPENed.

LOC is used to determine the current record number, that is, the number of the last record processed since the file was OPENed. It returns the record number accessed by the last GET or PUT statement.

LOC is also valid for sequential files. It returns the number of sectors (256-byte block) read from or written to the file since the file was OPENed.

Example

```
IF LOC(1)>55 THEN END
```

if the current record number is greater than 55, ends program execution.

Sample Program

```
1310 A$ = "WILLIAM WILSON"
1320 GET 1
1330 IF N$ = A$ THEN PRINT "FOUND IN RECORD"
     LOC(1): CLOSE: END
1340 GOTO 1320
```

This is a **portion** of a program. Elsewhere the file has been OPENed and FIELDed. N$ is a field variable. If N$ matches A$, the record number in which it was found is printed.

LOF

LOF(*buffer*)	**Function**

Returns the end-of-file record number.

Buffer is the number under which a file was OPENed.

This function tells you the number of the last record in a direct-access file.

Example

```
Y = LOF(5)
```

assigns the last record number to variable Y.

Sample Programs

During direct access to a pre-existing file, you often need a way to know when you've read the last valid record. LOF provides a way.

```
1540 OPEN "R", 1, "UNKNOWN/TXT", 255
1550 FIELD 1, 255 AS A$
1560 FOR I% = 1 TO LOF(1)     'LOF(1) = HIGHEST
1570 GET 1, I%                'RECORD NUM. TO BE
1580 PRINT A$                 'ACCESSED
1590 NEXT I%
1600 CLOSE
```

If you attempt to GET record numbers beyond the end-of-file, BASIC gives you an error.

When you want to add to the end of a file, LOF tells you where to start adding:

```
1600 I% = LOF(1) + 1     'HIGHEST EXISTING RECORD
1610 PUT 1, I%           'ADD NEXT RECORD
```

LOG

LOG(*number*)	**Function**

Computes the natural logarithm of *number*.

This is the inverse of the EXP function. The result is always in single precision.

Examples

```
PRINT LOG(3.14159)
```

prints the value 1.14472.

```
Z = 10 * LOG(Ps/P1)
```

performs the indicated calculation and assigns the value to Z.

Sample Program

This program demonstrates the use of LOG. It utilizes a formula taken from space communications research.

```
540 INPUT "DISTANCE SIGNAL MUST TRAVEL
    (MILES)"; D
550 INPUT "SIGNAL FREQUENCY (GIGAHERTZ)"; F
560 L = 96.58 + (20 * LOG(F)) + (20 * LOG(D))
570 PRINT "SIGNAL STRENGTH LOSS IN FREE SPACE
    IS" L "DECIBELS."
```

LPOS

LPOS(number) — **Function**

Returns the logical position of the line printer's print head within the line printer's buffer.

Number is a dummy argument.

This function does not necessarily give the physical position of the print head.

Example

```
100 IF LPOS(X)>60 THEN LPRINT
```

LPRINT, LPRINT USING

LPRINT ***data, . . .*** — **Statement**
LPRINT USING ***format; data, . . .***

Prints *data* on the printer.

See PRINT and PRINT USING for more information.

Examples

```
LPRINT (A * 2)/3
```

prints the value of expression (A * 2)/3 on the printer.

```
LPRINT TAB(50) "TABBED 50"
```

moves the line printer carriage to TAB position 50 and prints "TABBED 50". (Refer to the TAB function).

```
LPRINT USING "#####.#"; 2.17
```

sends the formatted value bbbb2.2 to the line printer.

LPRINT TAB

LPRINT TAB(*number*)	**Statement**

Works the same as PRINT TAB, except that it is for the printer.

Number can be in the range 1 to 255.

LSET

LSET *field name* = *data*	**Statement**

Sets *data* in a direct-access buffer *field name*.

Before using LSET, you must have used FIELD to set up buffer fields.

See also the chapter on "Disk Files", OPEN, CLOSE, FIELD, GET, PUT, and RSET.

Example

Suppose NM$ and AD$ have been defined as field names for a direct access file buffer. NM$ has a length of 18 characters; AD$ has a length of 25 characters. The statements

```
LSET NM$ = "JIM CRICKET, JR."
LSET AD$ = "2000 EAST PECAN ST."
```

set the data in the buffer as follows:

```
JIMbCRICKET,JR.bbb    2000bEASTbPECANbST.bbbbbb
```

Notice that filler blanks were placed to the right of the data strings in both cases. If we had used RSET statements instead of LSET, the filler spaces would have been placed to the left. This is the only difference between LSET and RSET.

If a string item is too large to fit in the specified buffer field, it is always truncated on the right. That is, the extra characters on the right are ignored. This applies to both LSET and RSET.

MEM

MEM	**Function**

Returns the amount of memory.

MEM performs the same function as FRE. It returns the number of unused and unprotected bytes in memory.

This function may be used in the immediate mode to see how much space a resident program occupies. It may also be used inside a program to avert "Out of memory" errors. MEM requires no argument.

See the FRE function for more information.

Example

```
PRINT MEM
```

Enter this command in the immediate mode (no line number is needed). The number returned indicates the amount of leftover memory; that is, memory not being used to store programs, variables, strings, the stack, or not reserved for object files.

Sample Program

```
1610 IF MEM < 80 THEN 1630
1620 DIM A(15)
1630 REM         PROGRAM CONTINUES HERE
```

If fewer than 80 bytes of memory are left, control switches to another part of the program. Otherwise, an array of 16 elements is created.

MERGE

MERGE *"filespec"*	**Statement**

Loads *filespec*, a BASIC program, and merges it with the program currently in memory.

Filespec specifies a BASIC file in ASCII format (a program saved with the A option). If *filespec* is a constant, it must be enclosed in quotes.

Program lines in the disk program are inserted into the resident program in sequential order. For example, suppose that three of the lines from the disk program are numbered 75, 85 and 90, and three of the lines from the current program are numbered 70, 80, and 90. When MERGE is used on the two programs, this portion of the new program will be numbered 70, 75, 80, 85, 90.

If line numbers on the disk program coincide with line numbers in the resident program, the disk program's lines replace the resident program's lines.

MERGE closes all files and clears all variables. Upon completion, BASIC returns to the command mode.

Example

Suppose you have a BASIC program on disk, PROG2/TXT (saved in ASCII), which you want to merge with the program you've been working on in memory. Then we use:

```
MERGE  "PROG2/TXT"
```

merges the two programs.

Sample Programs

MERGE provides a convenient means of putting program modules together. For example, an often-used set of BASIC subroutines can be tacked onto a variety of programs with this command.

Suppose the following program is in memory:

```
80 REM          MAIN PROGRAM
90 REM LINE NUMBER RESERVED FOR SUBROUTINE HOOK
100 REM         PROGRAM LINE
110 REM         PROGRAM LINE
120 REM         PROGRAM LINE
130 END
```

And suppose the following subroutine, SUB/TXT, is stored on disk in ASCII format:

```
90 GOSUB 1000 SUBROUTINE HOOK
1000 REM        BEGINNING OF SUBROUTINE
1010 REM        SUBROUTINE LINE
1020 REM        SUBROUTINE LINE
1030 REM        SUBROUTINE LINE
1040 RETURN
```

You can MERGE the subroutine with the main program with:

```
MERGE "SUB/TXT"
```

and the new program in memory is:

```
80   REM         MAIN PROGRAM
90   GOSUB  1000 SUBROUTINE HOOK
100  REM          PROGRAM LINE
110  REM          PROGRAM LINE
120  REM          PROGRAM LINE
130  END
1000 REM          BEGINNING OF SUBROUTINE
1010 REM          SUBROUTINE LINE
1020 REM          SUBROUTINE LINE
1030 REM          SUBROUTINE LINE
1040 RETURN
```

MID$

	Statement
MID$(*oldstring, position* [*,length*]) = *replacement string*	

Replaces a portion of an *oldstring* with *replacement string*.

Oldstring is the variable name of the string you want to change.

Position is a number specifying the position of the first character to be changed.

Length is a number specifying the number of characters to be replaced.

Replacement string is the string to replace a portion of *oldstring*.

The length of the resultant string is always the same as the original string. If *replacement string* is shorter than *length*, the entire replacement string is used.

Examples:

```
A$ = "LINCOLN"

MID$(A$, 3, 4) = "12345": PRINT A$
```

returns LI1234N.

```
MID$(A$, 5) = "Ø1": PRINT A$
```

returns LINCØ1N.

```
MID$(A$, 1, 3) = "***": PRINT A$
```

returns ***COLN.

MID$

MID$(*string, integer* [*,number*])	**Function**

Returns a substring of *string*, beginning at position *integer*.

If *integer* is greater than the number of characters in *string*, MID$ returns a null string.

Number is the number of characters in the substring. If omitted, BASIC returns all right most characters, beginning with the character at position *integer*.

Examples

If A$ = "WEATHERFORD" then

```
PRINT MID$(A$, 3, 2)
```

prints AT.

```
F$ = MID$(A$, 3)
```

puts ATHERFORD into F$.

Sample Program

```
200 INPUT "AREA CODE AND NUMBER
    (NNN-NNN-NNNN)"; PH$
210 EX$ = MID$(PH$, 5, 3)
220 PRINT "NUMBER IS IN THE " EX$ " EXCHANGE."
```

The first three digits of a local phone number are sometimes called the exchange of the number. This program looks at a complete phone number (area code, exchange, last four digits) and picks out the exchange of that number.

MKD$, MKI$, MKS$

Function

MKI$(*integer expression*)
MKS$(*single-precision expression*)
MKD$(*double-precision expression*)

Convert numeric values to string values.

Any numeric value that is placed in a direct file buffer with an LSET or RSET statement must be converted to a string.

These three functions are the inverse of CVD, CVI, and CVS. The byte values which make up the number are not changed; only one byte, the internal data-type specifier, is changed, so that numeric data can be placed in a string variable.

MKD$ returns an eight-byte string; MKI$ returns a two-byte string; and MKS$ returns a four-byte string.

Example

```
LSET AVG$ = MKS$(0.123)
```

Sample Program

```
1350 OPEN "D", 1, "TEST/DAT", 14
1360 FIELD 1, 2 AS I1$, 4 AS I2$, 8 AS I3$
1370 LSET I1$ = MKI$(3000)
1380 LSET I2$ = MKD$(3000.1)
1390 LSET I3$ = MKD$(3000.00001)
1400 PUT 1, 1
1410 CLOSE 1
```

For a program that retrieves the data from TEST/DAT, see CVD/CVI/CVS.

NAME

NAME *old filespec* AS *new filespec*	**Statement**

Renames *old filespec* as *new filespec*.

With this statement, the data in the file is left unchanged. The *new filespec* may not contain a password or drive specification.

Example

```
NAME "FILE" AS "FILE/OLD"
```

renames FILE as FILE/OLD.

```
NAME B$ AS A$
```

renames the file specified in B$ to the name specified in A$. Both A$ and B$ are unchanged.

NEW

NEW	**Statement**

Deletes the program currently in memory and clears all variables.

NEW displays a new (clear) screen and returns you to the command mode.

Example

```
NEW
```

OCT$

OCT$(*number*)	Function

Computes the octal value of *number*.

OCT$ returns a string which represents the octal value of *number*. The value returned is like any other string — it cannot be used in a numeric expression.

Examples

```
PRINT OCT$(30), OCT$(50), OCT$(90)
```

prints the following strings:

```
36        62        132
```

```
Y$ = OCT$(X/84)
```

Y$ is a string representation of the integer quotient X/84 to base 8.

ON ERROR GOTO

ON ERROR GOTO *line*	Statement

Transfers control to *line* if an error occurs.

This lets your program "recover" from an error and continue execution. (Normally, you have a particular type of error in mind when you use the ON ERROR GOTO statement).

ON ERROR GOTO has no effect unless it is executed before the error occurs. To disable it, execute an ON ERROR GOTO 0. If you use ON ERROR GOTO 0 inside an error-trapping routine, BASIC stops execution and prints an error message.

The error-handling routine must be terminated by a RESUME statement. See RESUME.

Example

```
10 ON ERROR GOTO 1500
```

branches program control to line 1500 if an error occurs anywhere after line 10.

For the use of ON ERROR GOTO in a program, see the sample program for ERROR.

ON . . . GOSUB

ON *expression* GOSUB *line*, . . .	**Statement**

Calls the subroutine at the *line* based on the value of *expression*.

Expression is a numeric expression between 0 and 255, inclusive. For example, if *expression*'s value is three, the third line number in the list is the destination of the branch.

If *expression's* value is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If *expression* is negative or greater than 255, an "Illegal function call" error occurs.

Example

```
ON Y GOSUB 1000, 2000, 3000
```

If Y = 1, the subroutine beginning at 1000 is called. If Y = 2, the subroutine at 2000 is called. If Y = 3, the subroutine at 3000 is called.

Sample Program

```
430 INPUT "CHOOSE 1, 2, OR 3" ; I
440 ON I GOSUB 500, 600, 700
450 END
500 PRINT "SUBROUTINE #1": RETURN
600 PRINT "SUBROUTINE #2": RETURN
700 PRINT "SUBROUTINE #3": RETURN
```

ON . . . GOTO

ON *expression* GOTO *line*, . . .	**Statement**

Goes to the *line* specified by the value of *expression*.

Expression is a numeric expression between 0 and 255.

This statement is very similar to ON . . . GOSUB. However, instead of branching to a subroutine, it branches control to another program line.

The value of *expression* determines to which line the program will branch. For example, if the value is four, the fourth line number in the list is the destination of the branch. If there is no fourth line number, control passes to the next statement in the program.

If the value of expression is negative or greater than 255, an "Illegal function call" error occurs. Any amount of line numbers may be included after GOTO.

Example

```
ON MI GOTO 150, 160, 170, 150, 180
```

tells BASIC to "Evaluate MI;
if the value of MI equals one then go to line 150;
if it equals two, then go to 160;
if it equals three, then go to 170;
if it equals four, then go to 150;
if it equals five, then go to 180;
if the value of MI doesn't equal any of the numbers one through five, advance to the next statement in the program".

OPEN

OPEN ***mode, buffer, "filespec"*** [***,record length***]	**Statement**

Opens a disk file.

Mode is a string expression whose first character is one of the following:

O for sequential output mode
I for sequential input mode
E for sequential output and extend mode
D or R for direct input/output mode

Buffer is an integer between 1 and 15. It specifies which area in memory you will use to access the file.

Filespec specifies a TRSDOS file.

If *Mode*=O, the *filespec* can also be a standard TRSDOS *devspec*, such as *PROR*CL.

Record length is an integer which sets the record length for direct-access files. The default is 256 bytes.

Once you have assigned a buffer to a file with the OPEN statement, that buffer cannot be used in another OPEN statement. You must first CLOSE the first file.

Examples

```
OPEN "D", 2, "DATA/BAS.SPECIAL"
```

opens the file DATA/BAS in direct-access mode, with the password SPECIAL. Buffer 2 is used. If DATA/BAS does not exist, it is created on the first non write-protected drive. The record length is 256 bytes.

```
OPEN "D", 5, "TEXT/BAS", 64
```

opens the file TEXT/BAS for direct access. Buffer 5 is used. The record length is 64. If this length does not match the record length assigned to TEXT/BAS when the file was originally OPENed, an error occurs.

```
OPEN "O", 7, "INV/CONT"
```

opens the sequential file "INV/CONT" for output. If "INV/CONT" does not exist, it is created. Information is written to the file sequentially, starting at the first byte. If the file does exist, any new information is written over the existing information; the file's previous contents are lost.

```
OPEN "E", 1, "LIST/EMP"
```

opens the file LIST/EMP and extends it by appending new data to the end of the file. If "LIST/EMP" does not exist, OPEN "E" works the same way as OPEN "O".

```
OPEN "I", 8, "MGT"
```

opens the sequential file "MGT" for sequential input. This enables you to retrieve information from the file (using INPUT# or LINE INPUT#). If "MGT" does not exist, a "File not found" error occurs.

See the chapter on "Disk Files" for programming information.

OPTION BASE

OPTION BASE *n*	**Statement**

Sets *n* as the minimum value for an array subscript.

N may be 1 or 0. The default is 0.

If you use this statement in a program, it must precede the DIM statement.

If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is one.

OUT

OUT *port, data byte*	Statement

Sends a *data byte* to a machine output *port*.

Port is an integer between 0 and 255. *Data byte* is also an integer between 0 to 255.

A port is an input/output location in memory. For information on assigned ports, see the *Model 4/4P Technical Reference Manual*.

Example

```
OUT 32,100
```

sends 100 to port 32.

PEEK

PEEK(*memory location*)	Function

Returns a byte from *memory location*.

The *memory location* must be in the range −32768 to 65535.

The value returned is an integer between 0 and 255. (For the interpretation of a negative value of *memory location*, see the statement VARPTR).

PEEK is the complementary function of the statement POKE.

Example

```
A = PEEK(&H5A00)
```

POKE

POKE *memory location, data byte* **Statement**

Writes *data byte* into *memory location.*

Both *memory location* and *data byte* must be integers. *Memory location* must be in the range −32768 to 65535.

POKE is the complementary statement of PEEK. The argument to PEEK is a memory location from which a byte is to be read.

PEEK and POKE are useful for storing data efficiently, loading assembly-language subroutines, and passing arguments (or results) to and from assembly-language subroutines.

For more information, see the *Model 4/4P Technical Reference Manual.*

Example

```
10 POKE &H5A00, &HFF
```

POS

POS(*number*) **Function**

Returns the position of the cursor.

Number is a dummy argument.

POS returns a number from 1 to 80 indicating the current cursor-column position on the display.

Example

```
PRINT TAB(40) POS(0)
```

prints 40. The PRINT TAB statement moves the cursor to position 40, therefore, POS(0) returns the value 40. (However, since a blank is inserted before the "4" to accommodate the sign, the "4" is actually at position 41).

Sample Program

```
150 CLS
160 A$ = INKEY$
170 IF A$ = "" THEN 160
180 IF POS(X) >70 AND A$=CHR$(32) THEN
    A$=CHR$(13)
190 PRINT A$;
200 LPRINT A$;
210 GOTO 160
```

This program lets you use your printer as a typewriter (except that you cannot correct mistakes). Your computer keyboard is the typewriter keyboard. The program will keep watch at the end of a line so that no word is divided between two lines.

PRINT

PRINT ***data, . . .***	**Statement**

Prints numeric or string *data* on the display.

BASIC prints the values of the data items you list in this statement.

You may separate the data items by commas or semicolons. If you use commas, the cursor automatically advances to the next tab position before printing the next item. (BASIC divides each line into five tab positions, at columns 0, 16, 32, 48, and 64). If you use semicolons, it prints the items without any spaces between them.

BASIC prints positive numbers with a leading blank. It prints all numbers with a trailing blank. This is done even if the numbers are separated by a semicolon.

A semicolon or comma at the end of a line causes the next PRINT statement to begin printing where the last one left off. If no trailing punctuation is used with PRINT, the cursor drops down to the beginning of the next line.

Single-precision numbers with six or fewer digits that can be accurately represented in ordinary (rather than exponential) format, are printed in ordinary format. For example, 1E-7 is printed as .0000001; 1E-8 is printed as 1E-08.

Double-precision numbers with 16 or fewer digits that can be accurately represented in ordinary format, are printed using the ordinary format. For example, 1D-15 is printed as .000000000000001; 1D-16 is printed as 1D-16.

To insert strings into this statement, surround them with quotation marks.

Examples

```
PRINT "DO"; "NOT"; "LEAVE"; "SPACES";
"BETWEEN"; "THESE"; "WORDS"
```

prints on the display:
DONOTLEAVESPACESBETWEENTHESEWORDS

Sample Program

```
60 INPUT "ENTER THIS YEAR"; Y
70 INPUT "ENTER YOUR AGE";A
80 INPUT "ENTER A YEAR IN THE FUTURE";F
90 N = A + (F - Y)
100 PRINT "IN THE YEAR"F"YOU WILL BE"N"YEARS
    OLD"
RUN
```

Since F and N are positive numbers, PRINT inserts a space before and after them, therefore your display should look similar to this (depending on your input):

```
IN THE YEAR 2004 YOU WILL BE 46 YEARS OLD
```

If we had separated each expression in line 100 by a comma,

```
100 PRINT "IN THE YEAR",F,"YOU WILL
    BE",N,"YEARS OLD"
```

BASIC would move to the next tab position after printing each data item.

PRINT TAB

PRINT TAB(*number*)	**Statement**

Moves the cursor to the column specified by *number*.

Because you can use numeric expressions to specify a tab position, TAB is useful for creating tables, graphs of mathematical functions, and other such screen displays.

When entering the PRINT TAB statement, type the first parenthesis immediately after the word TAB.

Number can be in the range 1 to 80. If *number* is less than or equal to the current screen width, BASIC uses *number* itself as the tab position. If *number* is greater than the current screen width, BASIC performs division, as follows, to determine the tab position.

If *number* is:	then BASIC divides *number* by:
greater than the current screen width but less than or equal to 80	the current width, and uses the remainder as the tab position.
greater than 80 (the maximum screen width)	80, and uses the remainder as the tab position.

For example, suppose that the screen width is 80 and you enter this statement:

```
PRINT "NAME"; TAB(90); "AMOUNT"
```

BASIC divides 90 by 80. The remainder is 10; so, BASIC moves the cursor from column 5, where it is positioned after NAME is printed, to column 10. It then prints AMOUNT at column 10.

Suppose that you had entered this statement, instead:

```
PRINT "NAME"; TAB(84); "AMOUNT"
```

The remainder is 4, but the current cursor position (the position after NAME is printed) is at column 5. BASIC cannot move the cursor to the left on the current line; so, it moves it instead to column 4 of the next line, and prints AMOUNT there.

If the string you are printing is too long to fit on the current line, BASIC moves the string to the first column on the next line.

Sample Program

```
10 PRINT "NAME" TAB(25) "AMOUNT":PRINT
20 READ A$, B$
30 PRINT A$ TAB(25) B$
40 DATA "G.T.JONES","$25.00"
```

The display shows:

```
NAME                    AMOUNT

G.T.JONES               $25.00
```

PRINT USING

PRINT USING ***format; data item, . . .***	**Statement**

Prints *data items* using a *format* specified by you.

Format consists of one or more field specifiers enclosed in quotes, or a string variable which contains the field specifier(s).

Data item may be string and/or numeric value(s).

This statement is especially useful for printing report headings, accounting reports, checks, or any other documents which require a specific format.

With PRINT USING, you may use certain characters (field specifiers) to format the field. These field specifiers are described below. They are followed by sample program lines and their output to the screen.

Specifiers for String Fields:

! Print the first character in the string only.

PRINT USING "!"; "PERSONNEL"
P

\spaces\ Print 2+ n characters from the string. If you type the backslashes without any spaces, BASIC prints two characters; with one space, BASIC prints three characters, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified and padded with spaces on the right. To enter a backslash, press CLEAR /.

```
PRINT USING "\bbb\"; "PERSONNEL"
```
(three spaces between the backslashes)
```
PERSO
```

& Print the string without modifications.

```
10 A$="TAKE":B$="RACE"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
TRACE
```

Specifiers for Numeric Fields:

Print the same number of digit positions as number signs (#). If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces). Numbers are rounded as necessary. You may insert a decimal point at any position. In that case, the digits preceding the decimal point are always printed (as zero, if necessary).

If the number to be printed is larger than the specified numeric field, a percent sign (%) is printed in front of the number. If rounding the number exceeds the field, a percent sign is also printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22
```

If the number of digits specified exceeds 24, an "Illegal function call" occurs.

```
PRINT USING "##.##";.75
0.75
```

```
PRINT USING "###.##";876.567
876.57
```

+ Print the sign of the number. The plus sign may be typed at the beginning or at the end of the format string.

```
PRINT USING "+##.##    ";
-98.45,3.50,22.22,-.9
-98.45    +3.50    +22.22    -0.90
```

```
PRINT USING "##.##+    ";
-98.45,3.50,22.22,-.9
98.54-    3.50+    22.22+    0.90-
```

(Note the use of spaces at the end of a format string to separate printed values).

–	Print a negative sign after negative numbers (and a space after positive numbers). PRINT USING "###.#–"; –768.660 768.7-
**	Fill leading spaces with asterisks. The two asterisks also establish two more positions in the field. PRINT USING "**####"; 44.0 ****44
$$	Print a dollar sign immediately before the number. This specifies two more digit positions, one of which is the dollar sign. PRINT USING "$$##.##"; 112.7890 $112.79
**$	Fill leading spaces with asterisks and print a dollar sign immediately before the number. PRINT USING "**$##.##"; 8.333 ***$8.33
,	Print a comma before every third digit to the left of the decimal point. The comma establishes another digit position. PRINT USING "####,.##"; 1234.5 1,234.50
^^^^	Print in exponential format. The four exponent signs are placed after the digit position characters. To type the ^, press (CLEAR)(;). You may specify any decimal point position. PRINT USING ".####^^^^"; 888888 .8889E+06
_	Print next character as a literal character. PRINT USING "_!##.##_!";12.34 !12.34!

Sample Program

```
420 CLS: A$ = "**$##,######.## DOLLARS"
430 INPUT "WHAT IS YOUR FIRST NAME"; F$
440 INPUT "WHAT IS YOUR MIDDLE NAME"; M$
450 INPUT "WHAT IS YOUR LAST NAME"; L$
460 INPUT "ENTER AMOUNT PAYABLE"; P#
470 CLS : PRINT "PAY TO THE ORDER OF  ";
480 PRINT USING "!! !!  "; F$; "."; M$;".";
490 PRINT L$
500 PRINT :PRINT USING A$; P#
```

In line 480, each ! picks up the first character of one of the following strings (F$, ".", M$, and "." again). Notice the two spaces in "!!b̸!!b̸". These two spaces insert the appropriate spaces after the initials of the name (see below). Also notice the use of the variables A$ for format and P for item list in line 500. Any serious use of the PRINT USING statement would probably require the use of variables at least for item list rather than constants. (We've used constants in our examples for the sake of better illustration.)

When the program above is run, the output should look something like this:

```
WHAT IS YOUR FIRST NAME? JOHN
WHAT IS YOUR MIDDLE NAME? PAUL
WHAT IS YOUR LAST NAME? JONES
ENTER AMOUNT PAYABLE? 12345.6
PAY TO THE ORDER OF J. P. JONES

*****$12,435.60 DOLLARS
```

PRINT @

PRINT@ *location*, **PRINT@ (*row*, *column*),**	**Statement**

Specifies exactly where printing is to begin.

The *location* specified must be a number between 0 and 1919. It can also be a pair of numbers (*r*, *c*), where row is a number in the range 0 to 23 and column is a number in the range 0 to 79.

Whenever you instruct BASIC to PRINT @ the bottom line of the display, it generates an automatic line feed; everything on the display moves up one line. To suppress this automatic line feed, use a trailing semicolon at the end of the statement.

If the sum of length of the string you are printing plus the current cursor position is greater than 80 or the current screen width, BASIC prints the entire string on the next line. (It does not examine the individual characters of the string in making this decision.) If your strings contain control characters, you may want to disable this feature. See WIDTH.

Examples

```
PRINT @ (11,39), "*"
```

prints an asterisk in the middle of the display. The space between PRINT and @ is optional.

```
PRINT @ 0, "*"
```

prints an asterisk at the top left corner of the display.

PRINT#

PRINT *#buffer, item1, item2, . . .*	Statement

Prints data *items* in a sequential disk file.

Buffer is the buffer number used to OPEN the file for input.

When you first OPEN a file for sequential output, BASIC sets a pointer to the beginning of the file — that's where PRINT# starts printing the values of the *items*. At the end of each PRINT# operation, the pointer advances, so values are written in sequence.

A PRINT# statement creates a disk image similar to what a PRINT to the display creates on the screen. For this reason, make sure to delimit the data so that it will be input correctly from the disk.

PRINT# does not compress the data before writing it to disk. It writes an ASCII-coded image of the data.

Examples

```
If A = 123.45
PRINT #1,A
```

writes this nine-byte character sequence onto disk:

```
b123.45b carriage return
```

The punctuation in the PRINT list is very important. Unquoted commas and semicolons have the same effect as they do in regular PRINT statements to the display. For example, if A = 2300 and B = 1.303, then

```
PRINT #1, A,B
(ENTER)
```

writes the data on disk as

```
b 2300 bbbbbbbbbb 1.303b carriage return
```

The comma between A and B in the PRINT# list causes 10 extra spaces in the disk file. Generally you wouldn't want to use up disk space this way, so you should use semicolons instead of commas.

Files can be written in a carefully controlled format using PRINT# USING. You can also use this option to control how many characters of a value are written to disk.

For example, suppose A$ = "LUDWIG", B$ = "VON", and C$ = "BEETHOVEN". Then the statement

```
PRINT #1, USING".!.!\bb\";A$;B$;C$
```

would write the data in nickname form:

```
L.V.BEET
```

(In this case, we didn't want to add any explicit delimiters.) See PRINT USING for more information on the USING option.

PUT

PUT *buffer* [*,record*]	**Statement**

Puts a *record* in a direct-access disk file.

Buffer is the same buffer used to OPEN the file.

Record is the record number you want to PUT into the file. It is an integer between 1 and 65535. If omitted, the current record number is used.

This statement moves data from the buffer of a file into a specified place in the file.

If *record* is higher than the end-of-file record number, then *record* becomes the new end-of-file record number.

The first time you use PUT after OPENing a file, you must specify the *record*. The first time you access a file via a particular buffer, the next record is set equal to one. (The next record is the record whose number is one greater than the last record accessed.)

See the chapter on "Disk Files" for programming information.

```
PUT 1
```

writes the next record from buffer 1 to a direct-access file.

```
PUT 1, 25
```

writes record 25 from buffer 1 to a direct-access file.

RANDOM

RANDOM	**Function**

Reseeds the random number generator.

If your program uses the RND function, every time you load it, BASIC generates the same sequence of pseudorandom numbers. Therefore, you may want to put RANDOM at the beginning of the program. This will help ensure that you get a different sequence of pseudorandom numbers each time you run the program.

RANDOM needs to execute just once.

Sample Program

```
600 CLS : RANDOM
610 INPUT "PICK A NUMBER BETWEEN 1 AND 5"; A
620 B = RND(5)
630 IF A = B THEN 650
640 PRINT "YOU LOSE, THE ANSWER IS" B "--TRY
    AGAIN."
645 GOTO 610
650 PRINT "YOU PICKED THE RIGHT NUMBER -- YOU
    WIN": GOTO 610
```

READ

READ *variable, ...*	**Statement**

Reads values from a DATA statement and assigns them to *variables.*

BASIC assigns values from the DATA statement on a one-to-one basis. The first time READ is executed, the first value in the first DATA statement is used; the second time, the second value is used, and so on.

A single READ may access one or more DATA statements (each DATA statement is accessed in order), or several READs may access the same DATA statement.

The values read must agree with the variable types specified in list of variables, otherwise, a "Syntax error" occurs. If the number of variables in the READ statement exceeds the number of elements in the DATA statement(s), an "Out of data" error message is printed.

If the number of variables specified is lower than the number of elements in the DATA statement(s), subsequent READ statements begin reading data at the first unread element.

Example

```
READ T
```

reads a numeric value from a DATA statement and assigns it to variable "T".

Sample Program

This program illustrates a common application for the READ and DATA statements.

```
40 PRINT "NAME", "AGE"
50 READ N$
60 IF N$="END" THEN PRINT "END OF LIST": END
70 READ AGE
80 IF AGE<18 THEN PRINT N$, AGE
90 GOTO 50
100 DATA "SMITH, JOHN", 30, "ANDERS, T.M.", 20
110 DATA "JONES, BILL", 15, "DOE, SALLY", 21
120 DATA "COLLINS, W.P.", 17, "END"
```

REM

REM	Statement

Inserts a remark line in a program.

REM instructs the computer to ignore the rest of the program line. This allows you to insert remarks into your program for documentation. Then, when you look at a listing of your program, or someone else does, it will be easier to figure it out.

If REM is used in a multi-statement program line, it must be the last statement in the line.

You may use an apostrophe (') as an abbreviation for REM.

Sample Program

```
110 DIM V(20)
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
```

OR

```
110 DIM V(20)
120 FOR I=1 TO 20     'CALCULATE AVERAGE VELOCITY
130 SUM=SUM + V(I)
140 NEXT I
```

RENUM

Statement

RENUM [***new line***] [,[***line***] [,***increment***]]

Renumbers a program, starting at *line*, using *new line* as the first new line and *increment* for the new sequence.

If you omit *new line*, BASIC starts numbering at line 10. If you omit *line*, it renumbers the entire program. If you omit *increment*, it jumps 10 numbers between lines.

RENUM also changes all line number references appearing after GOTO, GOSUB, THEN, ELSE, ON . . . GOTO, ON . . . GOSUB, ON ERROR GOTO, RESUME, and ERL[relational operator].

If the program contains line number references that do not exist, RENUM does not change those line number references. We recommend that you save a copy of the program before you renumber it. If there are any non-existent line number references after you renumber the program they will be easier to correct.

Examples

```
RENUM
```

renumbers the entire resident program, incrementing by 10's. The new number of the first line will be 10.

```
RENUM 600, 5000, 100
```

renumbers all lines numbered from 5000 up. The first renumbered line will become 600, and an increment of 100 will be used between subsequent lines.

```
RENUM 10000, 1000
```

renumbers line 1000 and all higher-numbered lines. The first renumbered line will become line 10000. An increment of 10 will be used between subsequent line numbers.

```
RENUM 100, , 100
```

renumbers the entire program, starting with a new line number of 100, and incrementing by 100's. Notice that the commas must be retained even though the middle argument is gone.

Error Conditions

1. RENUM cannot be used to change the order of program lines. For example, if the original program has lines numbered 10, 20 and 30, then the command:

   ```
   RENUM 15, 30
   ```

   is illegal, since the result would be to move the third line of the program ahead of the second. In this case, an "Illegal function call" error occurs, and the original program is left unchanged.
2. RENUM will not create new line numbers greater than 65529. Instead, an "Illegal function call" error occurs, and the original program is left unchanged.
3. If an undefined line number is used inside you original program, RENUM prints a warning message, "Undefined line XXXX in YYYY", where XXXX is the original line number reference and YYYY is the original number of the line containing XXXX. Note that RENUM renumbers the program in spite of this warning message. It does not change the incorrect line number reference, but it does renumber YYYY, according to the parameters in your RENUM command.

RESTORE

RESTORE [*line*]	**Statement**

Restores a program's access to previously-read DATA statements.

This lets your program re-use the same DATA lines. If *line* is specified, the next READ statement accesses the first item in the specified DATA statement.

Sample Program

```
160 READ X$
170 RESTORE
180 READ Y$
190 PRINT X$, Y$
200 DATA FIRST ITEM, SECOND ITEM
```

When this program is run,

```
FIRST ITEM    SECOND ITEM
```

is printed on the display. Because of the RESTORE statement in line 170, the second READ statement starts over with the first DATA item.

RESUME

RESUME [*line*] **RESUME NEXT**	**Statement**

Resumes program execution after an error-handling routine.

RESUME without an argument and RESUME 0 both cause the computer to return to the statement in which the error occurred and re-execute the statement.

If your error routine did not correct the error, the error occurs again and the ON ERROR GOTO is taken.

RESUME *line* causes the computer to return to the specified line number.

RESUME NEXT causes the computer to return to the statement following the point at which the error occurred.

A RESUME that is not in an error-handling routine causes a "RESUME without error" message.

Examples

```
RESUME
```

if an error has occurred, this line transfers program control to the statement in which it occurred.

```
RESUME 10
```

if an error has occurred, transfers control to line 10.

Sample Program

```
10 ON ERROR GOTO 900
.
.
.
900 IF (ERR=230) AND(ERL=90) THEN PRINT "TRY
    AGAIN" : RESUME 80
```

RETURN

RETURN	Statement

Returns control to the line immediately following the most recently executed GOSUB.

If the program encounters a RETURN statement without execution of a matching GOSUB, an error occurs.

Sample Program

```
330 PRINT "THIS PROGRAM FINDS THE AREA OF A
    CIRCLE"
340 INPUT "TYPE IN A VALUE FOR THE RADIUS"; R
350 GOSUB 370
360 PRINT "AREA IS" ; A: END
370 A = 3.14 * R * R
380 RETURN
```

RIGHT$

RIGHT$(*string, number*)	Function

Returns the rightmost *number* characters of *string*.

RIGHT$ returns the last *number* characters of *string*. If LEN (string) is less than or equal to *number*, the entire string is returned.

Examples:

```
PRINT RIGHT$("WATERMELON", 5)
```

prints MELON

```
PRINT RIGHT$("MILKY WAY", 25)
```

prints MILKY WAY.

Sample Program

```
850 RESTORE : ON ERROR GOTO 880
860 READ COMPANY$
870 PRINT  RIGHT$(COMPANY$, 2), : GOTO 860
880 END
890 DATA "BECHMAN LUMBER COMPANY, SEATTLE, WA"
900 DATA "ED NORTON SEWER SERVICE, BROOKLYN, NY"
910 DATA "HAMMOND MANUFACTURING COMPANY,
    HAMMOND, IN"
```

This program prints the name of the state in which each company is located.

RND

RND(*number*)	**Function**

Generates a pseudorandom number between 0 and *number*.

Number must be greater than or equal to 0 and less than 32768.

RND produces a pseudorandom number using the current "seed" number. BASIC generates the seed internally, therefore, it is not accessible to the user. RND may be used to produce random numbers between 0 and 1, or random integers greater than 0, depending on the argument.

RND(0) returns a single-precision value between 0 and 1, RND(number) returns an integer between 1 and *number*. For example, RND(55) returns a pseudorandom integer between 1 and 55. RND(55.5) returns a pseudorandom number between 1 and 56 (the argument is rounded).

Examples

```
A = RND(2)
```

assigns A a value of 1 or 2.

```
A = RND(45)
```

assigns A a random integer between 1 and 45.

```
PRINT RND(0)
```

prints a decimal fraction between 0 and 1.

ROW

ROW(*number*)	Function

Returns the row position of the cursor.

Number is a dummy argument.

ROW finds the row in which the cursor is currently located and returns that row number. The 24 rows are numbered 0-23.

Examples

```
X = ROW(Y)
```

assigns the cursor's current row number to X.

Sample Program

When you type a key, the program below prints: the keyboard character, the cursor's row number and column number, and the character's ASCII code.

```
100 CLS
110 R=0: C=0
120 PRINT@(21,32), "ROW", "COLUMN"
130 X$ = INPUT$(1)
140 PRINT @(R,C), X$;
150 C=POS(0): R=ROW(0)
160 PRINT @ (22,32),R,C;
163 PRINT @ (23,32), STRING$(20,32);
165 PRINT @(23,32), "ASCII CODE IS
    "HEX$(ASC(X$));
170 PRINT @ (R,C),"";
180 GOTO 130
```

RSET

RSET ***field name = data***	**Statement**

Sets *data* in a direct-access buffer *field name*.

This statement is similar to LSET. The difference is that with RSET, data is right-justified in the buffer.

See LSET for details.

RUN

RUN [***line***] **RUN** ***filespec***[,R]	**Statement**

Runs a program.

RUN followed by a *line* or nothing at all simply executes the program in memory, starting at *line* or at the beginning of the program.

RUN followed by a *filespec* loads a program from disk and then runs it. Any resident BASIC program is replaced by the new program.

Option R leaves all previously OPEN files open. If omitted, BASIC closes all open files.

RUN automatically CLEARS all variables. However, it does not re-set the value of an ERL variable.

Examples

```
RUN
```

starts execution at lowest line number.

```
RUN 100
```

starts execution at line 100.

```
RUN "PROGRAM/A"
```

loads and executes PROGRAM/A.

```
RUN "EDITDATA", R
```

loads and executes EDITDATA, leaving OPEN files open.

SAVE

SAVE *"filespec"* [,**A**] [,**P**]	**Statement**

Saves a program in a disk file under *filespec*.

If *filespec* already exists, its contents will be lost as the file is re-created.

SAVE without the A option saves the program in a compressed format. This takes up less disk space. It also helps in performing SAVEs and LOADs faster. BASIC programs are stored in RAM using compressed format.

Using the A option causes the program to be saved in ASCII format. This takes up more disk space. However, the ASCII format allows you to MERGE this program later on. Also, data programs which will be read by other programs must usually be in ASCII.

For compressed-format programs, a useful convention is to use the extension BAS. For ASCII-format programs, use /TXT.

The P option protects the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it fails. The only operations that can be performed on a protected file are: RUN, LOAD, MERGE, and CHAIN.

Examples

```
SAVE  "FILE1/BAS.JOHNQDOE:3"
```

saves the resident BASIC program in compressed format. The file name is FILE1; the extension is /BAS; the password is JOHNQDOE. The file is placed on Drive 3.

```
SAVE "MATHPAK/TXT", A
```

saves the resident program in ASCII form, using the name MATHPAK/TXT, on the first non-write-protected drive.

SGN

SGN(*number*)	**Function**

Determines *number's* sign.

If *number* is a negative number, SGN returns −1. If *number* is a positive number, SGN returns 1. If *number* is zero, SGN returns 0.

Examples

```
Y = SGN(A * B)
```

determines what the sign of the expression A * B is, and passes the appropriate number (−1,0,1) to Y.

Sample Program

```
610 INPUT "ENTER A NUMBER"; X
620 ON SGN(X) + 2 GOTO 630, 640, 650
630 PRINT "NEGATIVE": END
640 PRINT "ZERO": END
650 PRINT "POSITIVE": END
```

SIN

SIN(*number*)	**Function**

Computes the sine of *number*.

Number must be in radians. To obtain the sine of *number* when *number* is in degrees, use SIN(number * .01745329). The result is always single precision.

Examples

```
PRINT SIN(7.96)
```

prints .994385.

Sample Program

```
660 INPUT "ANGLE IN DEGREES"; A
670 PRINT "SINE IS"; SIN (A * .01745329)
```

SOUND

SOUND *tone, duration*	**Statement**

Generates a sound with the *tone* and *duration* specified.

Tone is a digit between 0 and 7. It specifies the sound's frequency level. Zero specifies the lowest frequency level; seven specifies the highest.

Duration is an integer between 0 and 31. It specifies for how long the sound is to be generated. Zero specifies the shortest duration; 31 the longest.

While the SOUND statement is being executed, you cannot interrupt it. You can, however, stop BASIC as soon as SOUND finishes executing. To do this, hold down (BREAK).

This statement can be especially useful in educational applications. For example, you can have the computer respond with a sound if a user has answered a program's prompt incorrectly (or vice versa).

Sample Program

```
10 INPUT "IN HONOR OF WHOM WAS THE CONTINENT OF
   AMERICA NAMED"; A$
20 IF A$="AMERIGO VESPUCCI" THEN SOUND 7,2 ELSE
   GOTO 40
30 PRINT "THAT'S RIGHT!": END
40 SOUND 1,2 : PRINT "THE CORRECT ANSWER IS
   AMERIGO VESPUCCI"
```

SPACE$

SPACE$(*number*)	**Function**

Returns a string of *number* spaces.

Number must be in the range 0 to 255.

Example

```
PRINT "DESCRIPTION" SPACE$(4) "TYPE" SPACE$(9)
"QUANTITY"
```

prints DESCRIPTION, four spaces, TYPE, nine spaces, QUANTITY

Sample Program

```
920 PRINT "Here"
930 PRINT SPACE$(13) "is"
940 PRINT SPACE$(26) "an"
950 PRINT SPACE$(39) "example"
960 PRINT SPACE$(52) "of"
970 PRINT SPACE$(65) "SPACE$"
```

SPC

SPC(*number*)	Function

Prints a line of *number* blanks.

Number is in the range 0 to 255. SPC does not use string space. The left parenthesis must immediately follow SPC.

SPC may only be used with PRINT, LPRINT, or PRINT#.

Example

```
PRINT "HELLO" SPC(15) "THERE"
```

prints HELLO, 15 spaces, THERE

SQR

SQR(*number*)	Function

Calculates the square root of *number*.

The *number* must be greater than zero.

The result is always single precision.

Example

```
PRINT SQR(155.7)
```

prints 12.478.

Sample Program

```
680 INPUT "TOTAL RESISTANCE (OHMS)"; R
690 INPUT "TOTAL REACTANCE (OHMS)"; X
700 Z = SQR((R * R) + (X * X))
710 PRINT "TOTAL IMPEDANCE (OHMS) IS" Z
```

This program computes the total impedance for series circuits.

STOP

STOP	**Statement**

Stops program execution.

When a program encounters a STOP statement, it prints the message BREAK IN, followed by the line number that contains the STOP. STOP is primarily a debugging tool. During the break in execution, you can examine variables or change their values.

The CONT command resumes execution at the point it was halted. But if the program itself is altered during the break, CONT cannot be used.

Sample Program

```
2260 X = RND(10)
2270 STOP
2280 GOTO 2260
```

A random number between 1 and 10 is assigned to X, then program execution halts at line 2270. You can now examine the value X with PRINT X. Type CONT to start the cycle again.

STR$

STR$(*number*)	Function

Converts *number* into a string.

If *number* is positive, STR$ places a blank before the string.

While arithmetic operations may be performed on *number*, only string functions and operations may be performed on the string.

Example

```
S$ = STR$(X)
```

converts the number X into a string and stores it in S$.

Sample Program

```
10 A = 1.6 : B# = A : C# = VAL(STR$(A))
20 PRINT "REGULAR CONVERSION" TAB(40) "SPECIAL
   CONVERSION"
30 PRINT B# TAB(40) C#
```

STRING$

STRING$(*number,character*)	Function

Returns a string of *number characters*.

Number must be in the range 0 to 255.

Character is a string or an ASCII code. If you use a string constant, it must be enclosed in quotes. All the characters in the string will have either the ASCII code specified, or the first letter of the string specified.

STRING$ is useful for creating graphs or tables.

Examples:

```
B$ = STRING$(25, "X")
```

puts a string of 25 "X"s into B$.

```
PRINT STRING$(50, 10)
```

prints 50 blank lines on the display, since 10 is the ASCII code for a line feed.

Sample Program

```
1040 CLEAR 300
1050 INPUT "TYPE IN THREE NUMBERS BETWEEN 33
     AND 159"; N1, N2, N3
1060 CLS: FOR I = 1 TO 4: PRINT STRING$(20,
     N1): NEXT I
1070 FOR J = 1 TO 2: PRINT STRING$(40, N2):
     NEXT J
1080 PRINT STRING$(80, N3)
```

This program prints three strings. Each string has the character corresponding to one of the ASCII codes provided.

SWAP

SWAP *variable1, variable2*	Statement

Exchanges the values of two variables.

Variables of any type may be SWAPped (integer, single precision, double precision, string). However, both must be of the same type, otherwise, a "Type mismatch" error results.

Either or both of the variables may be elements of arrays. If one or both of the variables are non-array variables which have not been assigned values, an "Illegal Function Call" error results.

Example

```
SWAP F1#, F2#
```

swaps the contents of F1# and F2#. The contents of F2# are put into F1#, and the contents of F1# are put into F2#.

Sample Program

```
10 A$="ONE ":B$="ALL ":C$="FOR "
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
ONE FOR ALL
ALL FOR ONE
```

SYSTEM

SYSTEM [*"command"*]	**Statement**

Returns you to TRSDOS level.

Command tells the system to execute the specified TRSDOS command and immediately return to BASIC. Your program and variables are not affected. If *command* is a constant, it must be enclosed in quotes. You can specify only the TRSDOS library commands, not the utilities.

If you omit *command*, SYSTEM returns to the TRSDOS Ready mode. Your resident BASIC program is not retained in memory.

NOTE: You cannot call DEBUG from BASIC.

If an error occurs during the execution of *command*, BASIC either stops and displays an error message or transfers control to the ON ERROR routine. ERR$ will contain the TRSDOS error or BASIC error 70 (Command Aborted).

Some library commands have options that cannot be used in the SYSTEM command. For example, the CAT and DIR commands cannot sort filenames because of insufficient memory. Therefore, TRSDOS always uses SORT=NO; if you specify SORT=YES, TRSDOS ignores it.

Examples

```
SYSTEM
```

returns you to TRSDOS. Your resident BASIC program is lost.

```
SYSTEM "DIR"
```

runs the TRSDOS command, DIR (print directory), then returns to BASIC. Your resident BASIC program remains intact.

TAN

TAN(*number*)	Function

Computes the tangent of *number*.

Number must be in radians. To obtain the tangent of *number* when it is in degrees, use TAN (number * .01745329). The result is always single precision.

Examples

```
PRINT TAN(7.96)
```

prints −9.39702

Sample Program

```
720 INPUT "ANGLE IN DEGREES"; ANGLE
730 T = TAN(ANGLE * .01745329)
740 PRINT "TAN IS" T
```

TIME$

TIME$	Function

Returns the time of the day.

This function lets you use the time in a program.

The operator sets the time initially when TRSDOS is started up. When you request the time, TIME$ supplies it using this format:

```
14:47:18
```

which means 14 hours, 47 minutes and 18 seconds (24-hour clock).

To change the time, use the TRSDOS command, TIME. For example,

```
SYSTEM "TIME 10:15:00"
```

Example

```
A$ = TIME$
```

stores the current time in A$.

Sample Program

```
1130 SYSTEM "TIME 10:15:00"
1140 IF LEFT$(TIME$, 5) = "10:15" THEN PRINT
     "Time is 10:15 A.M.--time to pick up the
     mail." : END
1150 GOTO 1140
```

TROFF, TRON

TROFF **TRON**	**Statements**

Turn the "trace function" on/off.

The trace function lets you follow program flow. This is helpful for debugging and analyzing of the execution of a program.

Each time the program advances to a new line, TRON displays that line number inside a pair of brackets. TROFF turns the tracer off.

Sample Program

```
2290 TRON
2300 X = X * 3.14159
2310 TROFF
```

Lines 2290 and 2310 above might be helpful in assuring you that line 2300 is actually being executed, since each time it is executed [2300] is printed on the display.

After a program is debugged, the TRON and TROFF statements can be removed.

USR

USR[*digit*](*expression*)	**Function**

Calls a user's assembly-language subroutine identified with *digit* and passes *expression* to that subroutine.

The *digit* you specify must correspond to the *digit* supplied with the DEF USR statement for that routine. If *digit* is omitted, zero is assumed.

This function lets you call as many as 10 machine-language subroutines, then continue execution of your BASIC program. Subroutines must have been previously defined with DEF USR[digit] statements.

When BASIC encounters a USR call, it transfers control to the address defined in the DEF USR[digit] statement. (This address specifies the entry point to your machine-language subroutine.)

"Machine language" is the low-level language used internally by your computer. It consists of Z-80 microprocessor instructions. Machine-language subroutines are useful for special applications (things you can't do in BASIC) and for doing things very fast (like to "white-out" the display).

Writing such routines requires familiarity with assembly-language programming and with the Z-80 instruction set. There are books available on this subject; check your local Radio Shack or a book store.

Example

```
X = USR5(Y)
```

calls the machine-language routine USR5, previously defined in a DEF USR5 = *address* statement.

Passing arguments from BASIC to the subroutine:

Upon entry to a USR subroutine, the following register contents are set up (for notation, see the TRSDOS reference section in this manual):

A	=	Type of argument in USR[digit] reference A = 8 if argument is double-precision A = 4 if argument is single-precision

A = 2 if argument is integer
A = 3 if argument is string

HL	=	When the argument is a number, this register points to the argument storage area(ASA) described later.
DE	=	When the argument is a string, this register points to a string description, as follows: The first byte gives the length of the string. The next two bytes give the address where the string is stored: least significant byte (LSB) followed by most significant byte (MSB).

Description of Argument Storage Area (ASA) — for numeric values only.

For double-precision numbers:

ASA + 3	Exponent in 128-excess form, e.g., a value of 128 indicates a 0 exponent; a value of 66 indicates a −62 exponent. A value of 0 always indicates the number is zero.
ASA + 2	Highest seven bits of the mantissa with hidden (implied) leading one. Bit 7 is the sign of the number (0 positive, 1 negative), e.g., a value of X'84' indicates the number is negative and the MSB of the mantissa is X'84'. A value of X'04' indicates the number is positive and the MSB of the mantissa is X'84'.
ASA + 1	Next MSB of the mantissa.
ASA	Next MSB.
ASA - 1	Next MSB.
ASA - 2	Next MSB.
ASA - 3	Next MSB.
ASA - 4	Lowest eight bits of the mantissa.

For single-precision numbers:

ASA	LSB of the mantissa.
ASA + 1 through ASA + 3	Same as for double-precision numbers.

For integers:

ASA	LSB of the number
ASA + 1	MSB of the number. Together, the two bytes represent the number in signed, two's complement form.

Your routine can call BASIC's FRCINT routine to put the argument into HL in 16-bit, signed two's complement form. The address of FRCINT is stored in [X'2603', X'2604'].

For example, you can put the following code at the beginning of your subroutine:

```
FRCINT  EQU     2603H           ;CONVERTS USR ARGUMENT
                                ;TO INTEGER IN HL
        LD      HL,CTNU         ;(HL)=CONTINUATION
                                ;ADDRESS
        PUSH    HL              ;SAVE IT FOR RETURN
                                ;FROM FRCINT
        LD      HL,(FRCINT)     ;(HL)=FORCE INTEGER
                                ;ROUTINE
        JP      (HL)            ;DO FRCINT ROUTINE
```

Returning values from the subroutine to BASIC:

If the USR[digit] expression is a variable, you can modify its value by changing the ASA or string contents, as pointed to by HL or DE. For example, the statement:

```
X=USR1(A%)
```

transfers control to the USR1 subroutine, with HL pointing to the two-byte ASA for integer variable A%. Suppose you modify the contents of its storage area. When you do a RET instruction to return to BASIC, A% will have a new value, and X will be assigned this new value.

In general, USR[digit](expression) will return the same type of value as expression. However, you can use BASIC's MAKINT routine to return an integer value. The address of the MAKINT routine is stored at [X'2605',X'2606'].

For example, you might include the following code at the end of your program to return a value to BASIC:

```
MAKINT  EQU     2605H
        LD      HL,VAL          ;VAL IS THE VALUE TO
                                ;BE RETURNED.
        PUSH    HL              ;SAVE VALUE IN STACK
        LD      HL, (MAKINT)    ;RESTORE VAL INTO HL
        EX      (SP), HL        ;AND PUT MAKINT
                                ;INTO STACK
        RET
```

VAL

VAL(*string*)	**Function**

Calculates the numerical value of *string*.

VAL is the inverse of the STR$ function; it returns the number represented by the characters in a string argument. This number may be integer, single precision, or double precision, depending on the range of values and the rules used for typing all constants.

For example, if A$ = "12" and B$ = "34" then VAL(A$ + "." + B$) returns the value 12.34 and VAL(A$ + "E" + B$) returns the value 12E34, that is, 12 * 10^34.

VAL terminates its evaluation on the first character which has no meaning in a numeric value.

If the string is non-numeric or null, VAL returns a zero.

Examples

```
PRINT VAL("100 DOLLARS")
```

prints 100.

```
PRINT VAL("1234E5")
```

prints 1.234E+08.

```
B = VAL("3" + "*" + "2")
```

assigns the value 3 to B (the asterisk has no meaning in a numeric term).

Sample Program

```
10 READ NAMES$, CITY$, STATE$, ZIP$
20 IF VAL(ZIP$) < 90000 OR VAL(ZIP$) > 96699
   THEN PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$) > 90801 AND VAL(ZIP$) <= 90815
   THEN PRINT NAME$ TAB(25) "LONG BEACH"
```

VARPTR

VARPTR (*variable*) or **VARPTR (*#buffer*)**	**Function**

Returns the absolute memory address.

VARPTR can help you locate a value in memory. When used with *variable*, it returns the address of the first byte of data identified with *variable*.

When used with *buffer*, it returns the address of the file's data buffer.

If the *variable* you specify has not been assigned a value, an "Illegal Function Call" occurs. If you specify a *buffer* that was not allocated when loading BASIC, a "Bad file number" error occurs. (See Chapter 1 for information on how to load BASIC.)

VARPTR is used primarily to pass a value to a machine-language subroutine via USR[digit]. Since VARPTR returns an address which indicates where the value of a variable is stored, this address can be passed to a machine-language subroutine as the argument of USR; the subroutine can then extract the contents of the variable with the help of the address that was supplied to it.

If VARPTR returns a negative address, add it to 65536 to obtain the actual address.

If VARPTR(integer variable) returns address K:

Address K contains the least significant byte (LSB) of the 2-byte integer.

Address K + 1 contains the most significant byte (MSB) of the integer.

If VARPTR(single-precision variable) returns address K:

(K)*	= LSB of value
(K + 1)	= Next most significant byte(Next MSB)
(K + 2)	= MSB with hidden (implied) leading one. Most significant bit is the sign of the number
(K + 3)	= exponent of value excess 128(128 is added to the exponent).

If VARPTR(double-precision variable) returns K:

(K)	= LSB of value
(K + 1)	= Next MSB
(K + . . .)	= Next MSB
(K + 6)	= MSB with hidden (implied) leading one. Most significant bit is the sign of the number.
(K + 7)	= exponent of value excess 128 (128 is added to the exponent).

*(K) signifies "contents of address K"

For single and double-precision values, the number is stored in normalized exponential form, so that a decimal is assumed before the MSB. 128 is added to the exponent. Furthermore, the high bit of MSB is used as a sign bit. It is set to 0 if the number is positive or to 1 if the number is negative. See examples below.

If VARPTR(string variable) returns K:

(K)	= length of string
(K + 1)	= LSB of string value starting address
(K + 2)	= MSB of string value starting address

The address will probably be in high RAM where string storage space has been set aside. But, if your string variable is a constant (a string literal), then it will point to the area of memory where the program line with the constant is stored, in the program buffer area. Thus, program statements like A$="HELLO" do not use string storage space.

For all of the above variables, addresses (K-1) and (K-2) stores the TRS-80 Character Code for the variable name. Address (K-3) contains a descriptor code that tells the computer what the variable type is. Integer is 02; single precision is 04; double precision is 08; and string is 03.

VARPTR(array variable) returns the address for the first byte of that element in the array. The element consists of 2 bytes if it is an integer array; 3 bytes if it is a string array; 4 bytes if it is a single precision array; and 8 bytes if it is a double precision array.

The first element in the array is preceded by:

1. A sequence of two bytes per dimension, each two-byte pair indicating the "depth" of each respective dimension.
2. A single byte indicating the total number of dimensions in the array.
3. A two-byte pair indicating the total number of elements in the array.
4. A two-byte pair containing the ASCII-coded array name.

5. A one-byte type-descriptor(02 = Integer. 03=String. 04 = Single-Precision. 08 = Double-Precision).

Item 1 immediately precedes the first element. Item 2 precedes Item 1, and so on.

The elements of the array are stored sequentially with the first dimension-subscripts varying "fastest", then the second, etc.

Examples

A! = 2 is stored as follows:

2 = 10 Binary, normalized as .1E2 = .1 × 10 (to the second)
So exponent of A is 128+2 = 130 (called excess 128)
MSB of A is 10000000; however, the high bit is changed to zero since the value is positive(called hidden or implied leading one).

So A! is stored as

Exponent(K + 3)	MSB(K + 2)	Next MSB(K + 1)	LSB(K)
130	0	0	0

A!=.5 is stored as

Exponent(K + 3)	MSB(K + 2)	Next MSB(K + 1)	LSB(K)
128	128	0	0

A!=7 is stored as

Exponent(K + 3)	MSB(K + 2)	Next MSB(K + 1)	LSB(K)
131	96	0	0

A!=7:

Exponent(K + 3)	MSB(K + 2)	Next MSB(K + 1)	LSB(K)
131	224	0	0

Zero is stored as a zero-exponent. The other bytes are insignificant.

```
Y = USR1(VARPTR(number))
```

If number is an integer value, VARPTR(number) finds the address of the least significant byte of number. This address is passed to the subroutine, which in turn passes its result to Y.

WAIT

WAIT *port, integer1* [*,integer2*]	**Statement**

Suspends program execution until a machine input *port* develops a specified bit pattern. (A port is an input/output location.)

The data read at the port is exclusive OR'ed with *integer2*, then AND'ed with *integer1*. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If *integer2* is omitted, it is assumed to be zero.

It is possible to enter an infinite loop with the WAIT statement. In this case, you will have to manually restart the machine. To avoid this, WAIT must have the specified value at port number during some point in program execution.

For information on assigned ports, refer to the *Model 4/4P Technical Reference Manual*.

Example

```
100 WAIT 32,2
```

WHILE . . . . WEND

WHILE *expression*	**Statement**
.	
.	
.	
.	
{**loop statements**}	
.	
WEND	

Execute a series of statements in a loop as long as a given condition is true.

If *expression* is not zero (true), BASIC executes loop statements until it encounters a WEND. BASIC returns to the WHILE statement and checks *expression*. If it is still true, BASIC repeats the process. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND matches the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND causes a "WEND without WHILE" error.

Sample Program

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115 FLIPS=0
120 FOR I=1 TO J-1
130 IF A$(I)>A$(I+1)THEN SWAP A$(I), A$(I+1):
    FLIPS=1
140 NEXT I
150 WEND
```

This program sorts the elements in array A$. Control falls out of the WHILE loop when no more SWAPS are performed on line 130.

WIDTH

WIDTH [LPRINT] *size*	Statement

Sets the line width in number of characters for the display or line printer. If you omit the LPRINT option, BASIC sets the width at the screen.

size may be an integer in the range 15 to 255 that specifies the number of characters in a line. If you omit the LPRINT option, *size* can be 15 to 80 for the screen.

If *size* is 255, the printer width is infinite. That is, BASIC never inserts a carriage return. However, after printing the 255th character, LPOS and POS return a value of 0.

Note: If you are using the TRSDOS program FORMS/FLT, the printer width must be set at 255 for FORMS/FLT to function properly.

If you execute a WIDTH statement to change the width of the screen or printer, we recommend that you execute another WIDTH statement to restore the width to its default values before you exit BASIC.

When width is set less than the default values, BASIC issues a carriage return after printing every *size* character. BASIC does not separate the characters in a string unless the string is longer than the total width of the screen. For example:

```
10 WIDTH 18
20 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ";"1234567890"
RUN
```

The screen displays:

```
ABCDEFGHIJKLMNOPQR
STUVWXYZ1234567890
Ready
```

After printing 18 characters of the string, BASIC issues a carriage return and prints the remaining characters in the first string. BASIC checks to see if there are enough positions left in the current line to print the second string and prints the second string.

If a string contains less characters than the width but there are not enough positions remaining on the current line, BASIC prints the string on the next line. BASIC only separates the characters in a string when the cursor is in the first column. For example:

```
10 WIDTH 18
20 PRINT " ABCDEFGHIJKLMNOPQRSTUVWXYZ ";"
   1234567890 "
RUN
```

The screen displays:

Before printing the first string, BASIC checks to see if there are enough positions remaining on the current line to print the entire string. The string contains 28 characters and the screen width is only 18. If the cursor is in Column 1, BASIC prints that portion of the string that will fit on the line, issues a carriage return and prints the remaining characters in the string on the next line. BASIC checks to see if there are enough positions left on the line to print the second string. There are only nine positions left on the current line and the second string contains 12 characters. BASIC issues a carriage return before printing the second string.

Example

```
10 WIDTH 20
20 PRINT "Marion Rich";"1002 Easy Street";
   "Arlington, TX  76013"
```

displays:

WRITE

WRITE [***data, . . .***] **Statement**

Writes *data* on the display.

WRITE prints the values of the data items you type. If *data* is omitted, BASIC prints a blank line. The *data* may be numeric and/or string. They must be separated by commas.

When the *data* are printed, each data item is separated from the last by a comma. Strings are delimited by quotation marks. After printing the last item on the list, BASIC inserts a carriage return.

Example

```
10 D=95:B=76:V$="GOOD BYE"
20 WRITE D, B, V$
RUN
 95, 76, "GOOD BYE"
Ready
```

WRITE#

WRITE ***#buffer, data, . . .***	**Statement**

Writes *data* to a sequential-access file.

Buffer must be the number used to OPEN the file.

The *data* you enter may be numeric or string expressions.

WRITE# inserts commas between the data items as they are written to disk. It delimits strings with quotation marks. Therefore, it is not necessary to put explicit delimiters between the data.

The items on *data* must be separated by commas.

WRITE# inserts a carriage return after writing the last data item to disk.

For example, if

```
A$="MICROCOMPUTER" and B$="NEWS"
```

the statement

```
WRITE #1, A$,B$
```

writes the following image to disk:

```
"MICROCOMPUTER","NEWS"
```

Part III/ Appendices

Appendix A/ Job Control Language

The TRSDOS Job Control Language (JCL) is one of the most powerful features of TRSDOS. It consists of:

- TRSDOS commands
- Macros
- Special symbols

You can use JCL to make your computer more "user friendly." That is, you can write JCL programs that perform a variety of functions, such as FORMAT and BACKUP, and have TRSDOS execute these functions when the user types in one command line.

If you have read the entries on the BUILD and DO commands, you know how to create a JCL file composed of TRSDOS commands. You can make this file more powerful by utilizing macros and other features of JCL. This section describes how.

The steps for creating and using a JCL file are:

1. Create a JCL file consisting of TRSDOS commands, macros, or special symbols. You can do this with the BUILD command, SCRIPSIT, or a BASIC program.

2. Execute the JCL file with the DO command. This causes the JCL processor to:

   - Take control of the keyboard (for line input)
   - Read a line in the DO file exactly as if it came from the keyboard
   - Return control of the keyboard to the user when it reaches the last line.

The following sections give complete information on all the JCL features:

- Simple JCL Execution
- Simple JCL Compiling
- Advanced JCL Compiling

Simple JCL Execution

This section lists the execution macros and gives examples on how to create and run a JCL file.

Creating a JCL File

A JCL file contains characters normally available from the keyboard (ASCII characters).

There are several ways to create a JCL file: the BUILD library command lets you create or extend a JCL file, but it does not let you edit an existing file. You can create and edit a JCL file with a BASIC program. A word processing system, such as SCRIPSIT, will also let you create or edit a JCL file.

Restrictions of JCL

- A JCL file line cannot be longer than 79 characters. Depending on the JCL method used (execute only or compile), JCL either ignores all characters after the 79th or aborts the processing entirely.
- Any program or utility with unpredictable prompts will not function properly when run from a JCL file.
- Any program or utility which requires removing the system disk causes the JCL to abort.
- You cannot execute certain TRSDOS library commands and utilities from a JCL file. The commands and utilities NOT valid from a JCL file are : certain BACKUP commands, BUILD, certain CONV commands. all (X) commands, DEBUG, certain PURGE commands, SYSGEN, and SYSTEM (SYSTEM=) command.
- As a general rule, you should not use a library command or utility when you specify the QUERY parameter.

Table 1/ Execution Macros

Macro Group	Group Description	Macros	Macro Description
Execution Comment		.Comment	Displays a comment on the screen during execution. Comments are written to SYSTEM/JCL.
Termination Macros	Terminate execution.	//ABORT	Stops execution, displays "Job aborted". Returns to TRSDOS or BASIC Ready.
		//EXIT	Stops execution, displays "Job done". Returns to TRSDOS or BASIC Ready.
		//STOP	Stops execution. Returns control to the user program.
Pause Delay Macros	Provide special functions.	//PAUSE	Suspends execution and displays a message.
		//DELAY	Suspends execution and displays a message for a specified amount of time.
		//WAIT	Suspends execution depending upon the setting of the system clock.
		//SLEEP	Suspends execution for a predetermined amount of time.
Alert Macros	Provide video and audio alerts.	//FLASH	Flashes a message on the screen a specified number of times.
		//ALERT	Provides an audible signal to the operator.
Keyboard Macros	Accept keyboard input.	//KEYIN	Selects predefined blocks of JCL lines.
		//INPUT	Inputs a line of information from the keyboard.

JCL Execution Macros

A macro is a pre-defined JCL instruction. //ABORT is an example of a macro symbol. Macro symbols must start at the first character position in the line. **An execution macro cannot be the first line in a JCL file.**

The JCL execution macros are:

//ABORT

Use this macro to exit a JCL procedure (if an error is encountered) and return to the program that initiated the DO command.

Your system returns you to the calling program if your JCL processing logic detects an error. The following message:

```
Job aborted
```

is displayed when an error is encountered.

//ALERT [(]*tone,silence,tone,silence, . . .*[)]

Use this macro to produce tones to the operator. //ALERT can generate up to eight different tones using the sound generator inside the computer.

You could use this macro to signify the end of a large JCL procedure. It could also be used during the execution of a procedure to bring attention to a specific process.

Tone is controlled by a number ranging from 0 - 7, with 7 producing the lowest tone and 0 producing the highest tone.

The tone is followed by a period of silence which you select with a second number ranging from 0 - 7, with 7 producing the longest period of silence and 0 producing the shortest period of silence. Tone and silence must be entered as number pairs (for example, "1,0"). You can enter as many number pairs as can fit on one line.

You can repeat the tone-silence sequence by enclosing the entire string in parentheses. The sequence keeps repeating until you press (ENTER), which continues execution of the JCL. Pressing (BREAK) aborts the JCL.

Any value entered (for tone or silence) is used in its modulo 8 form. That is, if you enter the number 8, a zero value is assumed. For example, the value 10 produces the tone assigned to 2.

//DELAY *duration*

The //DELAY macro provides a definite timed pause with execution automatically continuing at the end of the delay. The actual delay will be approximately 0.1 second per count. The count ranges from 1 to 256. Thus, a delay of from 0.1 second to 25.6 seconds is possible.

You could use the //DELAY macro to suspend execution long enough for you to make sure the printer is ready to print.

The execution time of a //DELAY macro will vary slightly according to the speed TRSDOS is running under (FAST or SLOW). See the SYSTEM library command.

//EXIT

Use this macro to halt execution of JCL processing and return to the program that initiated the DO command.

If you do not enter a termination macro in a JCL file, the JCL processing terminates when it reaches the end of the file (as if //EXIT were the last line in the JCL file). The following message is displayed:

```
Job done
```

This message indicates a normal conclusion of the JCL file.

You should use //EXIT if the conclusion of the JCL file also represents the conclusion of the job that is running. So, //EXIT can be used to conclude a program that does not require any more keyboard input, and needs to return to TRSDOS Ready or BASIC Ready after it finishes.

To conclude a program that requires additional keyboard input, use the //STOP macro. Using the //EXIT macro would terminate the program.

//FLASH [***duration***] ***message***

This macro flashes *message* on and off the video screen. *duration* is the number of times the *message* will flash and can be any number from 0 to 255. If *duration* is not specified, the message flashes 256 times. The message is any comment that you want displayed (up to 72 characters).

//KEYIN [***comment string***]

Use this macro to prompt for a single character entry (0 - 9), with the entire //KEYIN line being displayed.

During execution, press the appropriate character (0 - 9) to select the corresponding execution block in a JCL file. There can be up to ten execution blocks in a JCL file, each tagged with // and a number 0 - 9.

Do not use //KEYIN to enter data at execution time. If you do need to enter data at execution time, use the //INPUT macro.

//INPUT [***message string***]

Use this macro to input a line from the keyboard during JCL execution. With this macro, control of the keyboard is temporarily

returned to the operator. Now, any command can be typed on the keyboard and then passes to the system.

The number of characters allowed in the input line depends on where the JCL execution is when the //INPUT is encountered. For example, if the JCL is executing at the TRSDOS Ready level, then you can enter up to 80 characters, the same as for a normal TRSDOS command. If the //INPUT is encountered after going into BASIC, then you can enter up to 255 characters.

When you use the //INPUT macro, you should exercise some caution to assure that the command typed in is valid at the level it will be executed. For example, if you enter a program name incorrectly, the error message "Program not found" is displayed and the JCL execution aborts.

//**PAUSE** [***message string***]

When this macro is encountered in an executing JCL file, it is displayed on the screen along with a message. You can use the message to inform the operator why the pause was ordered. Press (ENTER) to resume JCL execution, or press (BREAK) to abort the JCL.

The //DELAY, //WAIT, and //SLEEP macros are similar to the //PAUSE macro, and are used to give JCL execution a specific delay period.

//**SLEEP** ***hh:mm:ss***

Use this macro to put the system "to sleep" for the amount of time you specify.

//SLEEP adds the specified time to the current system time and waits until that time to begin or resume execution.

Suppose you have two programs that begin execution every morning at 10 o'clock and each program runs for two hours. You could execute the first program and have the //SLEEP macro "halt" execution of the second program for an hour lunch break. After the system "sleeps" for the specified hour, the second program is executed.

//**STOP**

Use this macro to halt execution of a JCL file and return keyboard control to the applications program that requests additional keyboard input.

If you do not use the //STOP macro, you automatically return to TRSDOS Ready or BASIC Ready. You can also use the //ABORT and //EXIT macros to force an end to the JCL execution and return to TRSDOS Ready or BASIC Ready.

//WAIT *hh:mm:ss*

The //WAIT macro is similar to //DELAY, except that the length of the delay depends on the setting of the system clock.

The //WAIT macro puts the entire system in a "sleep" state until the system clock matches the time you specified.

You can set the system clock with the TIME library command. You can also set the time from a JCL file by using a direct execution of the TIME library command, or with the //INPUT macro. Set the clock in the format hh:mm:ss.

Examples

The easiest JCL file to understand is one containing only commands.

Use the BUILD command to create the following JCL file named START/JCL:

```
DEVICE
FREE
```

If you issue a DO = START command (see the DO library command) your computer displays the device table, lists free space information about all enabled drives, and returns to TRSDOS Ready.

Because an execution macro cannot be the first line in a JCL file, you could use an execution comment to display an informative message as the JCL file begins to execute. An execution comment begins with a period, which must be in the first character position of the line. You could label START/JCL as follows:

```
. This program executes the DEVICE and FREE
commands.
DEVICE
FREE
//EXIT
```

This comment describing the file's purpose is displayed when the JCL executes. Notice that we added the termination macro //EXIT.

You can use the //PAUSE macro in START/JCL as follows:

```
. This program executes the DEVICE and FREE
library commands.
//PAUSE Be sure the correct disk is in Drive Ø!
DEVICE
FREE
//EXIT
```

This example suspends the JCL before DEVICE executes, so you can be sure that the correct disk is in Drive 0. Press (ENTER) to continue the JCL.

You can use the //DELAY macro if you want to display an informative message to the operator. For example:

```
. BE SURE THAT THE PRINTER IS TURNED ON!
//DELAY 50
DEVICE (P)
FREE (P)
//EXIT
```

This example displays the above informative message and delays execution for approximately 5 seconds. After the delay, it executes DEVICE and FREE.

If you want your system to execute START/JCL at a certain time of the day, use the //WAIT macro as follows:

```
. This program runs at 2:15 a.m.
//WAIT 02:15:00
DEVICE
FREE
//EXIT
```

This example displays the comment and then waits until the system clock matches the time of 02:15:00 specified in the //WAIT macro. It would then execute DEVICE and FREE, and return to TRSDOS Ready.

```
. This program runs after a two-hour pause.
//SLEEP 02:00:00
DEVICE
FREE
//EXIT
```

This example displays the comment and then "sleeps" for two hours. It then executes DEVICE and FREE, and returns to TRSDOS Ready.

To use the //FLASH macro, modify START/JCL as follows:

```
. This program executes the DEVICE and FREE
commands.
DEVICE
//FLASH 10 Starting execution of FREE
FREE
//EXIT
```

After DEVICE executes, the //FLASH line is displayed. It flashes on and off 10 times, as specified by the duration count. You can press (ENTER) to stop the flash and proceed to the next line. Pressing (BREAK) while the message is flashing aborts the JCL and displays the message "Job Aborted".

You can modify START/JCL to show several uses of //ALERT:

```
. This program shows several uses of //ALERT.
. TURN TO PAGE 4 AT THE TONE.
//ALERT 0,0,1,5,0,2
. PRESS ENTER TO BEGIN EXECUTION.
//ALERT (1,0,7,0)
DEVICE
FREE
//EXIT
```

The first tone tells you when to turn to page 4. The second tone repeats until you press (ENTER) to continue execution of the program or (BREAK) to abort the JCL.

The next example shows how you could build a menu using execution comments to display different program choices. Using the //KEYIN macro lets you press a single key to execute the desired program.

```
. START/JCL
. Program 1 is FREE :0
. Program 2 is FREE
. Program 3 is DEVICE
//KEYIN Select program, 1 - 3
//1
FREE :0
//EXIT
//2
FREE
//EXIT
//3
DEVICE
//EXIT
///
```

There are two new macros used in this example. They are //*number* and ///.

//*number* is used to start a block of lines that corresponds to a value selected with the //KEYIN macro. This block extends until the next //*number* or to the ///.

/// (the triple slash) is used to mark the end of all //*number* blocks. JCL stops looking for a match as soon as it encounters a ///. Execution continues with the following line.

In the above example, pressing 1, 2, or 3 selects the corresponding block of lines and runs the appropriate command. If you press a key other than 1, 2, or 3, all three //*number* blocks are ignored, and execution continues with the line after the ///.

The lines following the /// could contain other command options or an //ABORT macro to abort the JCL. One possible option could be to let the operator type in his own command.

Consider the following rewrite of START/JCL that uses the //INPUT macro to let the operator type in his own command:

```
. START/JCL
. Program 1 is FREE :0
. Program 2 is FREE
. Program 3 is DEVICE
//KEYIN Select program, 1 - 3
//1
FREE :0
//EXIT
//2
FREE
//EXIT
//3
DEVICE
//EXIT
///
//INPUT Enter your own choice of command.
//EXIT
```

Now, if you press a key other than 1, 2, or 3 for the //KEYIN, the //INPUT line is displayed.

You can also enter information directly into the system at the JCL level. For example, the //WAIT macro description mentions that you can set the time for the system clock in the middle of a JCL file.

The following example prompts you to enter the TIME library command to set the system clock. After you input the time, the //WAIT macro pauses execution of the JCL file until the clock matches 02:15:00 and then continues execution.

```
. This program runs at 2:15 a.m.
//INPUT Enter the TIME command using HH:MM:SS format
//WAIT 02:15:00
DEVICE
FREE
//EXIT
```

JCL Compiling

The previous section explained how to create and use execute JCL files. This section describes some basic functions of the JCL compiler and shows practical examples of JCL files.

While an execute JCL file is useful, you need to use the compile phase of JCL for extra features. These extra features are explained in four parts:

- Compilation Description and Terms
- Conditional Decisions
- Substitution Fields
- Combining Files

Compilation Description and Terms

You can compile and/or execute any JCL file using the DO library command. If your JCL file contains only execution comments, commands, or execution macros, then you can completely skip the compile phase (using the "=" control character with the DO command).

If, however, your JCL file contains "tokens" or labels, or must make logical decisions, then you must compile the file before executing it.

The compile phase reads in the JCL file line by line, checking for directly executable lines, keyboard responses, and execution macros. The compile phase also evaluates any compilation statements and writes *all* resultant lines to a file called SYSTEM/JCL. After the compile phase completes, control is normally passed to the execution phase, which executes the SYSTEM/JCL file.

As stated earlier, the JCL works by substituting lines in a file for keyboard entries. However, when you compile a JCL file, it can contain more than just a series of executable commands. (After the compile phase is completed, however, the SYSTEM/JCL file *does* contain only executable lines.)

You can include the following statements in a JCL file you intend to compile:

- Directly executable commands (DIR, BASIC, etc.)
- Pre-arranged keyboard responses
- JCL execution macros (listed in Table 1)
- JCL conditional macros (listed in Table 2)
- Labels
- Another JCL file. When a JCL file "calls" another JCL file, TRSDOS transfers control to the called file and doesn't return control to the calling file.

We use several terms when discussing JCL compilation. They are:

1. Token

   A token is a string of up to 8 alphanumeric characters. You can use both upper and lower case letters. Note: There is NO difference between upper and lower case letters for any JCL macro, token, or label.

You can use a token (1) as a true/false switch for logical decisions (see the //IF macro), and (2) as a character string value in substitution fields (see the SUBSTITUTION FIELDS section).

2. Logical operator

   The simple logical operators are:

   AND (represented by the ampersand symbol "&")
   OR (represented by the plus symbol "+")
   NOT (represented by the minus symbol "–")

3. Label

   A JCL label is used to define the start of a JCL procedure, which allows many small JCL procedures to be combined into one large file. The format for a label is:

   ```
   @label
   ```

   The *label* can be up to 8 alphanumeric characters long.

Table 2/ Conditional Macros

Macro Group	Group Description	Macros	Macro Description
Compilation Comment		//.Comment	Acts like a visual log of the JCL file because they are displayed during compilation. These comments are not written to SYSTEM/JCL.

Logical Macros	Define conditional "blocks."	//IF
		//END
		//ELSE
			Defines the start of a conditional block.
			Defines the end of a conditional block.
			Defines the alternative to a false //IF.

Higher Order Logical Macros	Provide for higher conditional logic statements.	//SET
		//RESET
		//ASSIGN
			Gives a token a logical true value.
			Gives a token a logical false value.
			Sets a token's value to true and assigns a character string value to the token.

Termination Macro		//QUIT	Aborts JCL compiling if an invalid condition is detected.
Merge Macro		//INCLUDE	Merges together two or more JCL files during compilation.

Conditional Decisions

Using //IF, //END, //ELSE

The logical compilation macros (//IF, //END, and //ELSE) are used to establish logical "blocks" in a JCL file. When a JCL file is being compiled, these blocks are evaluated as either true or false.

The //IF macro followed by a token determines if the block is true or false.

To set a token true, specify it on the DO command line. To set a token false, do NOT specify it on the DO command line.

These JCL macros produce the following results:

1) If token is true . . .

```
//IF token
Include these lines.
//END
```

2) If token is false . . .

```
//IF token
Ignore these lines.
//END
```

3) If token is false, perform the alternative . . .

```
//IF token
Ignore these lines.
//ELSE
Include these lines.
//END
```

With this type of logical decision capability, you can create a JCL file and then pick a course of action by typing in a "DO *filespec*" command with different tokens.

Examples

Consider the following JCL file named START/JCL.

```
. START/JCL for program start-up
SET *FF to FORMS/FLT
FILTER *PR *FF
//IF PR1
FORMS (CHARS=80)
//ELSE
FORMS (CHARS=132)
//END
```

Assume that these are the first lines in a JCL file that begins execution of an applications program.

To make the //IF PR1 test as true, issue the following DO command:

```
DO START (PR1) (ENTER)
```

The 80 characters per line mode is selected.

If (PR1) is not specified on the DO command line, then the //IF test is false and the 132 characters per line is selected.

Using //SET and //RESET

JCL provides the //SET and //RESET macros to reduce the number of tokens in the DO command line.

One basic use for //SET is to let one token set the value of another. For example:

```
//IF KI
//SET P1
//END
```

This JCL file specifies that if KI is true, then P1 is set to a true condition also.

Suppose that the token P2 is already SET and you want to give it a new value. Consider this example:

```
//IF KI
//RESET P2
//END
```

This JCL file specifies that if KI is true, then P2 is reset to a false condition.

Consider the JCL file named MENU/JCL:

```
. MENU/JCL, revision 1
SET *FF TO FORMS/FLT
FILTER *PR *FF
//IF P1
//RESET P2
FORMS (CHARS=80)
//ELSE
//SET P2
FORMS (CHARS=132)
//END
```

If you issue either one of the following commands:

```
DO MENU (P1) (ENTER)
DO MENU (P1,P2) (ENTER)
```

the //IF macro tests P1 as true; therefore P2 is reset to false, and the *FF and 80-character mode are applied.

If you issue either one of the following commands:

```
DO MENU (ENTER)
DO MENU (P2) (ENTER)
```

the //IF macro tests false so the //ELSE macro sets P2 to true and the 132-character mode is applied.

As previously mentioned, the //SET macro can be used to reduce the number of tokens that have to be entered on the DO command line. Consider the following SYSOPT/JCL example:

```
. Establish TRSDOS system options
//IF ALL
//SET COMM
//SET PR
//SET SET
//SET SRES
//END
//IF KIALL
//SET COMM
//SET SET
//END
//IF COMM
set *cl to com/dvr
setcom (word=8)
//END
//IF PR
set *ff to forms/flt
filter *pr *ff
forms (chars=80)
//END
//IF SET
setki (rate=7)
//END
//IF SRES
system (sysres=2)
system (sysres=3)
system (sysres=10)
//END
```

This example shows how many different TRSDOS options can be established with a JCL file. The way it is structured, you can choose any or all of the options.

If you did not use //SET, you would have to enter four separate tokens on the DO command line to establish all of the options, as follows:

```
DO SYSOPT/JCL (COMM,PR,SET,SRES) (ENTER)
```

If you specify "ALL" in the DO command line, COMM, PR, SET, and SRES are set to true conditions.

If you specify "KIALL" in the DO command line, COMM and SET are set to true conditions.

Notice the use of upper and lower case. As stated earlier, either upper or lower case letters can be used in any JCL macro, token, or label. This is also true when the line is a TRSDOS command, as are the lower case lines in this example.

You can improve the readability of a JCL file by using upper case for macros and lower case for executable lines, such as TRSDOS commands, or vice versa.

Using //ASSIGN

JCL provides the //ASSIGN macro to set a token's logical value true, and to assign a character string value to a token.

The syntax for the //ASSIGN macro is:

```
//ASSIGN token=character string
```

character string can consist of up to 32 characters. Any character on the keyboard is allowed except a double-quotation mark (").

Error Conditions

Any time you use //ASSIGN, there must be at least one character assigned as a value or the compiling aborts.

Examples

In any of the previous examples that used the //SET macro, the //ASSIGN macro could have been substituted. The character string value assigned to the token has no effect on the JCL logic.

In the following example, if the token A is true, the tokens P1, KI, and PR are all set to true. This example assigns character string values to the tokens.

```
.TEST/JCL
//IF A
//ASSIGN P1=PROGRAM/BAS
//ASSIGN KI=ALL
//ASSIGN PR=80
//END
```

Using //. Comment and //QUIT

Compilation comments (//. Comment) are not written to the SYSTEM/JCL file. They are displayed on the screen as they are encountered during compilation. Thus, they act as a visual status log of the compile.

The //QUIT macro aborts the compilation stage if the JCL detects an invalid condition. This macro lets you make sure all needed tokens are entered before any execution takes place.

Examples

```
. START/JCL
set *ff to forms/flt
filter *pr *ff
forms (lines=60)
//IF KI
setki (rate=5)
//ELSE
//. RATE was not set!
//QUIT
//END
//EXIT
```

If this JCL file is compiled without the token KI being entered on the DO command line, the screen display shows:

```
//. RATE was not set!
//QUIT
```

No actual lines are executed from the SYSTEM/JCL file, because the compile phase was aborted before completion. The compilation comment tells the operator why the abort took place.

If you substitute //ABORT for //QUIT in the previous example and then compile the JCL file without the token KI, the following lines result:

```
//. RATE was not set!
. START/JCL
set *ff to forms/flt
filter *pr *ff
forms (lines=60)
Job aborted
```

The comment line is displayed as the file is being compiled. However, since //ABORT is an execution macro, the SYSTEM/JCL file finishes compiling and then executes until it reaches the //ABORT line! The //QUIT macro should be used in such a case rather than the //ABORT.

Substitution Fields

One of the most powerful features of the JCL is its ability to substitute and concatenate (add together) character strings to create executable lines.

A substitution field is created by placing pound signs (#) around a token. When the file is compiled, this substitution token is replaced with its current value, either assigned on the DO command line or with the //ASSIGN macro.

Examples

```
. TEST/JCL
set *ff to forms/flt
filter *pr *ff
forms (chars=#C#)
basic
run"#P1#"
//STOP
```

This example uses two substitution fields: one in the FORMS command line representing the number of characters, and one in the RUN command line.

If you issue the DO command:

```
DO TEST (C=132,P1=PROGRAM1) (ENTER)
```

the lines written to the SYSTEM/JCL file are:

```
. TEST/JCL
set *ff to forms/flt
filter *pr *ff
forms (chars=132)
basic
run"PROGRAM1"
//STOP
```

The compile phase substitutes the character string value of the tokens into the actual command line!

The length of the replacement string does not have to be equal to the length of the token name between the # signs.

To reduce the number of tokens needed on the DO command line, and to increase the program options at the same time, use the //ASSIGN macro as follows:

```
. TEST/JCL
//ASSIGN c=80
//ASSIGN p1=program1
//IF num2
//ASSIGN c=132
//ASSIGN p1=program2
//END
set *ff to forms/flt
filter *pr *ff
forms (chars=#C#)
basic
run"#P1#"
//STOP
```

Specifying NUM2 overrides the 80-character printer filter and PROGRAM1 defaults. The values of C and P1 are automatically set with the //ASSIGN tokens inside the //IF conditional block.

Another use for substitution fields is replacing drive numbers.

The following example shows how a FORMAT and BACKUP JCL file can be structured:

```
. FB/JCL, FORMAT with BACKUP
//PAUSE Insert disk to format in drive #D#
format :#D# (name="data1",q=n,ABS)
backup :#S# :#D#
//EXIT
```

The token D represents the destination drive, and the token S represents the source drive.

If you enter the command:

```
DO FB/JCL (S=1,D=2) (ENTER)
```

the system pauses and prompts you to insert a disk in Drive 2.

Press (ENTER) and the JCL file continues. It formats the disk in Drive 2, and then it executes the backup command with Drive 1 as the source drive and Drive 2 as the destination drive.

The substitution fields can be used in message lines and comments as well as in executable command lines.

Be careful when you want to display a single "#" in a comment or message. Consider the following example:

```
//PAUSE Insert a disk in drive #1
```

If the JCL file were executed only, this line would be properly displayed. However, if the JCL were compiled, an error would occur. For this line to be properly displayed in a compiled JCL, it would have to be written as:

```
//PAUSE Insert a disk in drive ##1
```

Another practical use for substitution fields is copying password protected files from one drive to another.

```
. MOVE/JCL file transfer
copy program1.#P#:0 :#D#
copy program2.#P#:0 :#D#
copy program3.#P#:0 :#D#
copy program4.#P#:0 :#D#
//EXIT
```

In this example, a group of files is copied from Drive 0 to a drive specified in the DO command. Also, you have to supply the proper

password for the copies to work. If you specify the wrong password, an error is displayed and the JCL aborts.

Substitution fields can also be concatenated, or added together, to create new fields. For example:

```
. ADD/JCL
copy #F#/#E#:0 :1
copy #F1#/#E#:0 :1
//EXIT
```

This example uses two substitution fields, one for the filename and one for the extension.

If you issue the DO command:

```
DO ADD (F=SORT,E=CMD,F1=SORT1) (ENTER)
```

the following SYSTEM/JCL file results after compiling:

```
. ADD/JCL
copy SORT/CMD:0 :1
copy SORT1/CMD:0 :1
//EXIT
```

As in previous examples, the //IF and //ASSIGN macros could be used to allow a single token to select the F, F1, and E tokens.

Combining Files

Most of the JCL examples in the previous sections have been very short. In a practical operating environment, this is often the case. However, each of these small files is taking up the minimum disk allocation of one gran and using one directory entry.

To combine small files and save disk space, use the Label feature of JCL. You can also use the //INCLUDE macro to duplicate a JCL file inside of another JCL file, without having to retype the lines.

Using //INCLUDE

The //INCLUDE macro is used to merge together two or more JCL files during the compile phase. The syntax is:

```
//INCLUDE filespec
```

filespec is a JCL file.

This command is similar to specifying the filespec in a DO command line. However, you cannot enter tokens or other information after the filespec.

If you need to pass tokens to the included program, they will have to be established in the program that is doing the //INCLUDE.

Error Conditions

An //INCLUDE macro CANNOT be the last line in a JCL file. If it is, an "End of File Encountered" error occurs, and the JCL aborts.

Examples

This example shows two JCL files and the results of the compile phase. The two JCL files are:

```
. TEST1/JCL
. comment line 1
//INCLUDE TEST2
. comment line 2
//EXIT
```

```
. TEST2/JCL
. This comment is included
```

If you issue the command

```
DO TEST1 (ENTER)
```

the following SYSTEM/JCL file is produced:

```
. TEST1/JCL
. comment line 1
. TEST2/JCL
. This comment is included
. comment line 2
//EXIT
```

The compiling starts with the file named in the DO command line. As soon as the //INCLUDE is reached, all lines in the second JCL file are processed, and then the compiling returns to the rest of the original file.

There is no limit to the number of non-nested //INCLUDE macros you can use, other than having enough disk space for the resulting SYSTEM/JCL file.

Using JCL Labels

The LABEL feature of JCL allows you to permanently merge together many small JCL procedures into one large file, and then access those procedures individually. This saves disk space and directory entry slots.

Examples

```
. TEST/JCL label example
@FIRST
. this is the first procedure
//exit
@SECOND
. this is the next procedure
@THIRD
. this is the last procedure
```

This file contains three labels. To select any procedure, specify the label on the DO command line.

The following rules determine how much of a labeled JCL file is included in the compile phase:

1) If no label is specified on the DO command line, all lines from the beginning of the file up to the first label are compiled.

2) If a label is specified, compiling includes all lines from the specified label until the next label or the end of the file is reached.

DOing the TEST/JCL file using the @FIRST label would write the comment ". this is the first procedure" and the //EXIT macro to the SYSTEM/JCL file for execution. Specifying either of the other labels would include only the appropriate single comment line.

If you compiled the file without specifying a label in the DO command, only the initial execution comment ". TEST/JCL label example" would be written in SYSTEM/JCL.

There is no limit to the size of a labeled procedure. They can range from one to as many lines as you can fit on your disk. The only requirement is that a JCL file containing labels must be compiled.

When you use labels in a JCL file, we recommend that you start the file with a comment line or some executable line other than a label.

Suppose @FIRST is the first line in the following file:

```
@FIRST
. Print this comment
```

If you issued a DO command for this file without specifying the @FIRST label, the compiling phase would receive the first line, see that it is a label, and quit. Since the compile is complete, the SYSTEM/JCL file would be executed! And since nothing was written to SYSTEM/JCL, its old contents are not erased. In other words, whatever lines had been compiled to the SYSTEM/JCL file from a previous DO command would now be executed.

Advanced JCL Compiling

The previous section on JCL compiling described the basic uses of tokens and compilation macros. If you do not understand the JCL Compiling section, please re-read it. If you actually type in and try the examples, you will get a better understanding of how to structure a JCL file for compiling.

This section describes additional features and shows different ways to accomplish logical decision branching. These additional features are explained in four parts:

- Using the Logical Operators
- Using Nested //IF Macros
- Using Nested //INCLUDE Macros
- Using the Special % Symbol

Using the Logical Operators

The logical operators used with the //IF macro (AND, OR, and NOT) specify the type of logical testing, and they are represented as follows:

AND — ampersand (&)
OR — plus sign (+)
NOT — minus sign (–)

All previous examples of //IF tested the logical truth or falseness of a token. You can accomplish more complex and efficient testing by using the logical operators.

Consider the following series of examples using the tokens A and B:

```
//IF -A
. include these lines if A is not specified
//END
```

By using NOT (–), you can see if a token is false, which provides an alternative method to select a block of lines for compiling.

```
//IF A+B
. include these lines if A or B is specified
//END

//IF A&B
. include these lines if A and B are specified
//END
```

These examples show how multiple tokens may be tested in a single //IF statement. The first example is true if either A OR B is true. The second example is true only if both A AND B are true.

You can use any combination of logical operators in an //IF statement. The following rules apply:

- The expressions are evaluated from left to right.
- Do not use parentheses because they abort the JCL compiling.
- All logical operators have the same priority.

You can combine the logical operators to test almost any arrangement of tokens. You can combine the logical operators to set up default conditions and to check for missing tokens, as the following examples demonstrate.

```
. CHECK/JCL            . CHECK1/JCL
//IF -S                //IF -S+-D
//ASSIGN S=0           //. You MUST enter S and D!
//END                  //QUIT
//IF -D                //END
//ASSIGN D=2
//END
```

The CHECK example tests S and D individually, and assigns them default values if they were not true (that is, if they were not specified in the DO command line).

The CHECK1 example is structured so that both S and D must be true (specified on the DO command line), or the JCL compiling aborts.

Using Nested //IF Macros

By definition, a conditional block begins with an //IF and concludes with an //END.

When the //IF evaluates true, the lines between the //IF and the //END or an //ELSE (if one exists) are compiled. It is also possible to include other //IF - //END blocks within the main conditional block (called nesting).

The //ELSE macro provides an alternative course of action in case an //IF evaluates false. It is also possible to have more //IF - //END statements following the //ELSE. Refer to the following examples:

```
. TEST/JCL
//IF A
. comment 1
//ELSE
//IF B
. comment 2
//END        (ends the //IF B statement)
//END        (ends the //IF A statement)
```

If A evaluates true, comment 1 is written out, and the //ELSE is ignored. If A is false, B is tested. The comment 2 is written out only if B is true. Notice the two //END macros. There must be one //END for every //IF.

You can document your own JCL files in the same way that we have documented these examples.

Documenting //END macros increases the readability of the files, especially when you edit a file that you have created some weeks (or months) previously.

```
//IF A
. Comment A
//IF B
. Comment B
//IF C
. Comment C
//END          (ends Third IF)
//END          (ends Second IF)
. Comment D
//END          (ends First IF)
```

If the first //IF is false, all lines up to the corresponding //END are ignored.

If the first //IF is true, Comment A and Comment D are written to SYSTEM/JCL.

If //IF B is true, Comment B is also written to SYSTEM/JCL. If B is false, all lines up to the corresponding //END are ignored.

The only time //IF C is considered is if both A and B test true. If C is true, Comments A through D are written to SYSTEM/JCL.

Although not shown in the example, you can use the logical operators when nesting //IFs.

Using Nested //INCLUDE Macros

When you use the //INCLUDE macro, the included file can also contain another //INCLUDE macro. This is called nesting. The following rules apply:

- The maximum nest level is five active //INCLUDE macros.
- An //INCLUDE macro cannot be the last line in a JCL file.

Example

The following example uses three files to show how the lines in nested //INCLUDE files are processed:

```
//. NEST0/JCL
. nested procedure example (Nest 0)
//INCLUDE nest1
. this is the end of the primary JCL (Nest 0)
//EXIT

//. NEST1/JCL
. this is the first nest (Nest 1)
//INCLUDE nest2
. this is the end of the first nest (Nest 1)

//. NEST2/JCL
. this is the second nest (Nest 2)
```

If you save these JCL files as NEST0/JCL, NEST1/JCL, and NEST2/JCL and then compile and execute NEST0/JCL, the following SYSTEM/JCL results:

```
//. NEST0/JCL
//. NEST1/JCL
//. NEST2/JCL
. nested procedure example (Nest 0)
. this is the first nest (Nest 1)
. this is the second nest (Nest 2)
. this is the end of the first nest (Nest 1)
. this is the end of the primary JCL (Nest 0)
```

The //INCLUDE macro can be used to compile a large JCL procedure from a series of smaller JCL routines. If the finished SYSTEM/JCL file is a procedure that will be executed many times, you can easily save it by copying SYSTEM/JCL to a file with another name.

Using the Special % Symbol

The % symbol is used to pass character values (in hex) to the system as though they came from the keyboard. The syntax is:

%character value

Below are some valid values and their results:

Hex Value	Result
09	Position to next tab stop (every 8 columns)
0A	Linefeed
1F	Clear screen

The value of any printable character can also be used, although control characters (characters with a value less than hex 20) are generally used. (See Appendix C for a list of characters, values, and actions performed on the video display.)

Examples

You should place the clear screen character at the start of a line. For example:

```
%1F//PAUSE Insert disk in drive 1, press ENTER
```

clears the screen and displays the JCL line in the top left corner of the screen.

The tab and linefeed characters, used to position comments or lines on the screen, should always be placed AFTER the period in the comment line or the macro in an executable line. For example:

```
.%09%09 This comment is positioned at the
second tab stop.
//PAUSE %0A%0A%0A This line appears 3 lines down
```

If you place the character BEFORE the period, TRSDOS does not recognize it as a comment line and the JCL aborts.

If you place the character AFTER the macro, the //PAUSE is displayed and the remaining message line is displayed 3 lines lower on the screen.

Using the tab and linefeed characters in this manner can sometimes help to improve the readability of the messages displayed during JCL execution.

Using TRSDOS JCL To Interface With Applications Programs

This appendix describes how to use JCL to start up and control your applications programs.

Two languages are discussed: BASIC and Z-80 assembly.

Interfacing With BASIC

A JCL file is the perfect method to interface between the operating system and the BASIC language. JCL can be used to create procedures that require only the inserting of a diskette to start up a program. Additionally, you can utilize the features of JCL from within a BASIC program.

Examples

To use a JCL file to initiate an automatic start-up of a BASIC program, you can use the AUTO library command to execute a JCL file.

Assuming the JCL file is named BAS/JCL, issuing the command:

```
AUTO DO BAS/JCL (ENTER)
```

automatically executes the desired BASIC program every time the computer is booted with the AUTOed system disk.

In order to execute a BASIC program from a JCL file, lay out the JCL file as follows:

1. Establish any necessary drivers, filters, or other TRSDOS options.
2. Enter BASIC with any necessary parameters (such as memory size and number of files).
3. RUN the BASIC prcgram.

4. Terminate the JCL execution with //STOP (which leaves control with BASIC).

You can also enter a DO command directly from the TRSDOS Ready prompt to execute a BASIC program.

To execute a JCL file once you have entered BASIC, the command format is:

```
SYSTEM"DO filename"
```

This command can be typed in directly or entered as a BASIC program line.

Also, any JCL file called from BASIC should contain the //EXIT termination macro, so that control will return to TRSDOS Ready when the JCL file is completed.

For example, suppose you want to use the JCL //ALERT macro to inform you when a lengthy BASIC procedure has completed. Following the lines containing the BASIC procedure, you could have a BASIC program line such as:

```
1000 SYSTEM "DO = ALERT/JCL:0"
```

which executes the ALERT/JCL file:

```
. Your procedure is complete.  Press (ENTER) to
resume.
//ALERT (1,0,7,0)
BASIC
//STOP
```

When BASIC reaches line 1000, the JCL file ALERT/JCL is executed, sending a series of repeating tones out the tone generator.

You are notified that your BASIC procedure has completed. Pressing (ENTER) ends the JCL alert and returns you to BASIC.

There are two important points about this example. First, the comment line in the ALERT/JCL file is absolutely necessary, as a JCL file cannot start with an execution macro. Second, the "BASIC" statement will reload BASIC. If you want a particular program to be loaded and run, you can place its name on the command line or add the BASIC commands before the //STOP statement. The //STOP termination macro must be included to assure that keyboard control remains within BASIC.

Although the example demonstrates an execute only JCL file, you can also call compiled JCL procedures from BASIC. You can even construct a SYSTEM "DO *filespec* [(parameters)]" command using BASIC string substitution.

Any time you want to use a SYSTEM "DO *filespec*" command from BASIC to execute another BASIC program, you have to change the format of the command. To DO these types of JCL files from BASIC, use the commands:

```
SYSTEM (ENTER)
DO filespec [(parameters)](ENTER)
```

Using this format for the command assures that a proper exit is made before the new JCL file is started.

Controlling a BASIC program

In some cases, the prompts in a BASIC program can be answered with a line from a JCL file. This is true if the program uses the INPUT or LINEINPUT BASIC statement to take the input.

If the program uses the INKEY$ statement, response has to come from the keyboard rather than from a JCL file. If the program uses the proper input method, you can create a JCL for total hands-off operation as follows:

1. Run through the BASIC program, making a note of every prompt to be answered.
2. Create a JCL file to enter BASIC and run the program as explained above in the BAS/JCL example. Leave off the //STOP macro.
3. Add the responses to the prompts as lines in the JCL file.

Using this method provides automatic program execution. Terminating the JCL file depends on what needs to be done when the application program has completed.

If you want to run more programs, you could add the proper RUN"PROGRAM" line to the JCL file, followed by any required responses to program prompts.

If you want to return to the TRSDOS Ready mode, you could end the file with the //EXIT macro. If you want to return to the BASIC Ready mode, you could end the file with the //STOP macro.

Interfacing With Z-80 ASSEMBLY

It is very simple to interface an assembly language program with the DO processor. All programs that utilize the line input handler (identified as the @KEYIN supervisor call in the *Model 4/4P Technical Reference Manual*) are able to accept "keyboard" input from the JCL file, just as though you typed it in when the program ran.

This gives the capability of pre-arranging the responses to a program's requests for input, inserting the responses into the JCL file, initiating the procedure, then walking away from the machine while it goes about its business of running the entire job.

Keyboard input normally handled by the single-entry keyboard routines (@KBD, @KEY, and BASIC's INKEY$) continue to be requested from the keyboard at program run time and do not utilize the JCL file data for input requests.

Practical Examples Of TRSDOS JCL Files

It is virtually impossible to show all the many uses of JCL files.

We give you two examples of how you can make your day-to-day TRSDOS operations even more efficient using JCL files.

1)

This example shows how to SYSRES system modules using a JCL file. The modules to be resided are 2, 3, and 10. These modules have to be resident in memory to perform a backup by class between two non-system diskettes in a two-drive system.

The JCL file to SYSRES these modules may look something like this:

```
. BURES/JCL - JCL used to SYSRES modules 2, 3,
and 10
SYSTEM (SYSRES=2
SYSTEM (SYSRES=3
SYSTEM (SYSRES=10
. end of BURES/JCL
```

When executed, this JCL file causes the system modules 2, 3, and 10 to reside in high memory. Because this JCL uses no labels or compilation macros, the compilation phase can be skipped.

2)

This example shows how to back up a diskette using a JCL file.

A minimum of three drives are required. Drive 0 must contain a system diskette with the JCL file. Drive 1 contains the source diskette. Assume that the source diskette's name is MYDISK and its master password is PASSWORD. Also assume that it is 40 track, and double density. Drive 2 contains the destination diskette.

The JCL file to perform the backup may look something like this:

```
. DUPDISK/JCL - Disk duplication JCL
//PAUSE Source in 1, Dest. in 2, (ENTER) when ready
format :2 (name="mydisk",q=n,abs)
//PAUSE format ok? (ENTER) if yes, (BREAK) if no
backup :1 :2
. end of backup - will now restart JCL
do *
```

The second line of the JCL causes the computer to pause until the (ENTER) key is pressed. This allows you to insert the proper diskette into Drives 1 and 2. Once you insert the proper diskettes, press (ENTER) and the third line of the JCL is executed.

The format line passes the NAME parameter to the format utility. Note that the diskette name, and diskette password of the destination

diskette must be an exact match of the source disk. If they do not exactly match, the JCL aborts.

Also, note that the parameters Q=N and ABS are specified. Both are necessary. The Q=N parameter causes the computer to use the default of PASSWORD for the master password, by passing the "Master Password" prompt. The ABS parameter ensures that no prompt appears if the destination diskette contains data.

The pause after the format statement allows you to check whether or not the format is successful. If the destination diskette is properly formatted, press (ENTER) to continue the JCL.

After you press (ENTER) in response to the seconds pause, the backup takes place. When the backup completes, the comment line appears, and the DO * command executes. The command causes the SYSTEM/JCL file to execute. Realize that since this is a repeating JCL, the compilation phase cannot be skipped.

If tracks are locked out during the format, press (BREAK). Pressing (BREAK) aborts the JCL, and you have to restart the JCL file.

Important: Be aware that if BACKUP or FORMAT is being executed by a JCL file, the following rules apply:

1. If the backup is mirror image, the source and destination disk Disk ID's must be the same or the backup aborts.
2. Backups with the (X) parameter, single-drive backups, and backups with the (QUERY) parameter are not allowed.
3. Single-drive formats are not allowed.

Appendix B/ Hardware

The Keyboard Code Map

The keyboard code map shows the code that TRSDOS returns for each key, in each of the modes: control, shift, unshift, clear and control, clear and shift, clear and unshift.

For example, pressing (CLEAR), (SHIFT), and (1) at the same time returns the code X'A1'.

A program executing under TRSDOS — for example, BASIC — may translate some of these codes into other values. Consult the program's documentation for details.

(BREAK) Key Handling

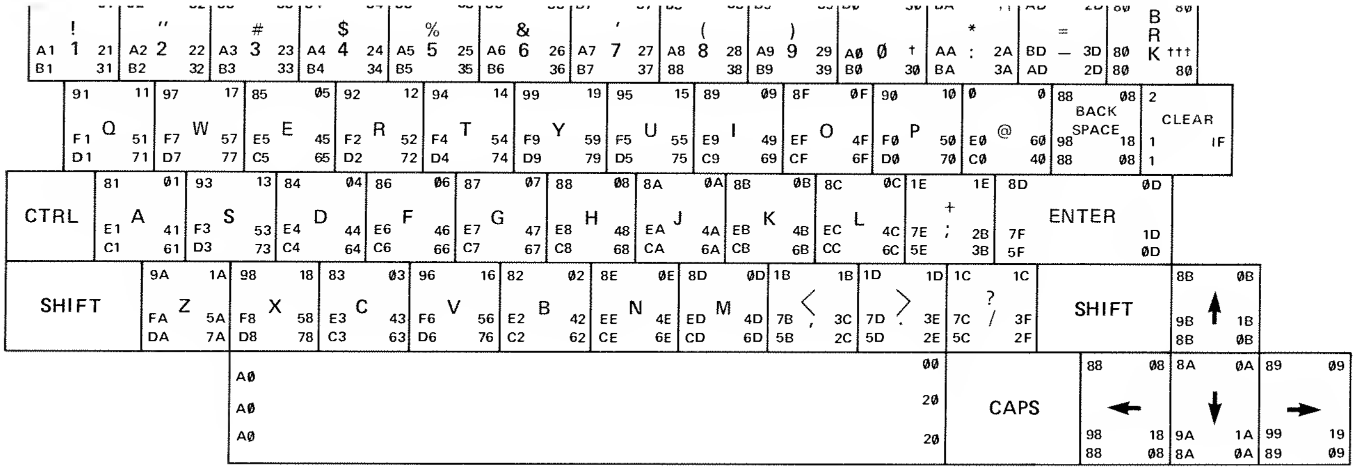
The (BREAK) key (X'80') is handled in different ways, depending on the settings of three system functions. The table below shows what happens for each combination of settings.

Break Enabled	Break Vector Set	Type-Ahead Enabled	
Y	N	Y	If characters are in the type-ahead buffer, then the buffer is emptied.* If the type-ahead buffer is empty, then a BREAK character (X'80') is placed in the buffer.*
Y	N	N	A BREAK character (X'80') is placed in the buffer.
Y	Y	Y	The type-ahead buffer is emptied of its contents (if any), and control is transferred to the address in the BREAK vector (see @BREAK SVC).*
Y	Y	N	Control is transferred to the address in the BREAK vector (see @BREAK SVC).
N	X	X	No action is taken and characters in the type-ahead buffer are not affected.

Y means that the function is on or enabled.
N means that the function is off or disabled
X means that the state of the function has no effect

Break is enabled with the SYSTEM (BREAK=ON) command (this is the default condition).
The break vector is set using the @BREAK SVC (normally off).
Type-ahead is enabled using the SYSTEM (TYPE=ON) command (this is the default condition).

* Because the BREAK key is checked far more frequently than other keys on the keyboard, it is possible for BREAK to be pressed after another key on the keyboard and yet be detected first.



LEGEND:

Clear and Control	• •	Control
Clear and Left Shift	• •	Shift
Clear and Unshift	• •	Unshift

NOTE: Pressing CONTROL, SHIFT, and @ at the same time generates and **EOF** (end of file) — — X'1C' with NZ return flag.

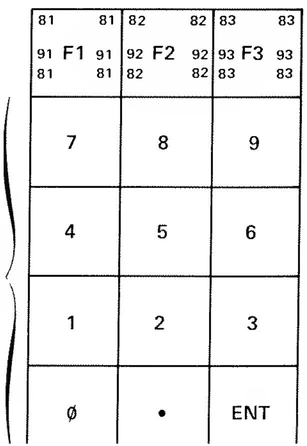
Whenever pressing CLEAR, SHIFT, and another key at the same time, be sure to use the left SHIFT key — not the right SHIFT key.

† Pressing SHIFT and Ø at the same time (or CAPS alone) turns the CAPS mode on or off.

†† Pressing CONTROL and : at the same time causes a screen print.

††† Pressing SHIFT and BREAK at the same time reselects the last drive.

Codes for these keys are the same as for the main keyboard.



Specifications

Your computer is a ROM/disk-based computer system with one major part:

- A display console/keyboard unit with two built-in, double-sided, double-density, floppy disk drives

The operating system software is loaded from ROM or an operating system disk in Drive 0 by a built-in read-only memory (ROM) "bootstrap" program.

Console

Processor

Your computer is a Z-80A based high-speed microprocessor with 64K or optional 128K of memory (disk system) or 16K of memory (cassette system)

The processor receives power-up and reset instructions from ROM.

The computer is compatible with existing Model III software.

Sound

The disk system can generate software-controlled tones, one at a time.

Video Display

Six Modes

- White on black or green on black (normal)
- Black on white or black on green (reversed)
- 64 characters by 16 lines Model III Mode
- 32 characters by 16 lines Model III Mode
- 80 characters by 24 lines 4D Mode
- 40 characters by 24 lines 4D Mode

Displayable Characters

- Full ASCII set
- 64 graphics characters

Keyboard

The keyboard has the standard typewriter keys, numeric keypad, and three function keys.

Three Modes

- Control
- Shift
- Caps
- Clear

Floppy Disk Drives

Minimum

Two built-in 5-¼-inch, double-sided floppy drives (disk system) or zero drives (cassette system)

Maximum

Two built-in 5-¼-inch, double-sided floppy disk drives and two external 5-¼-inch, floppy drives

Preventive Maintenance Interval

- Typical usage (3,000 power-on hours per year): Every 8,000 power-on hours
- Heavy usage (8,000 power-on hours per year): Every 5,000 power-on hours

Required Media

- Double-density double-sided, 5-1/4-inch floppy disks

The Data Transfer Rate is 250K bits per second.

Power Supply

Power Requirements

- 105-130 VAC, 60 Hz
- 240 VAC, 50 Hz (Australian / United Kingdom)
- 220 VAC, 50 Hz (European)
- Grounded outlet

Maximum Current Drain

- 1.7 Amps

Typical Current Drain

- 1.5 Amps

Operating Temperature

- 55 to 80 degrees Fahrenheit
- 13 to 27 degrees Centigrade

Peripheral Interfaces

Standard

- Floppy disk input/output channel for connection of one or two external floppy disk drives
- I/O bus for connection of hard disk and other peripherals
- Cassette I/O jack

Optional

- High-resolution graphics board
- Serial port RS-232C

Serial Interface

One Port:

- Allows asynchronous or synchronous transmission
- Conforms to the RS-232-C standard
- Uses the DB-25 connector on the back of the computer's display console

The DB-25 connector pin-outs and signals available are listed below:

Signal	Function	Pin#
PGND	Protective Ground	1
TD	Transmit Data	2
RD	Receive Data	3
RTS	Request to Send	4
CTS	Clear to Send	5
DSR	Data Set Ready	6
SGND	Signal Ground	7
CD	Carrier Detect	8
DTR	Data Terminal Ready	20
RI	Ring Indicator	22
STD†	Secondary Transmit Data	14
SUN†	Secondary Unassigned	18
SRTS†	Secondary Request to Send	19

† These signals are not used for secondary functions but are reserved for future use.

Parallel Interface

- Connection to a line printer via the 34-pin connector on the bottom of the computer's display console
- Eight data bits are output in parallel
- Eight data bits are input
- All levels are TTL compatible

The parallel printer pin-outs and signals available are listed below.

NOTE: If a signal name contains an asterisk (*), the signal is active-low.

Signal	Function	Pin #
STROBE*	1.5 microseconds pulse to clock the data from processor to printer	1
DATA 0	Bit 0 (lsb) of output data byte	3
DATA 1	Bit 1 of output data byte	5
DATA 2	Bit 2 of output data byte	7
DATA 3	Bit 3 of output data byte	9
DATA 4	Bit 4 of output data byte	11
DATA 5	Bit 5 of output data byte	13
DATA 6	Bit 6 of output data byte	15
DATA 7	Bit 7 (msb) of output data byte	17
BUSY	Input to compute from printer, high indicates busy	21
PAPER EMPTY	Input to computer from printer, high indicates no paper — If the printer doesn't provide this, the signal is forced low	23
BUSY* †	Inverse of BUSY (Pin 2)	25
FAULT*	Input to computer from printer, low indicates fault (paper empty, ribbon out, printer off-line, and so on)	28
GROUND	Common signal ground	2,4,6,8,10 12,14,16, 18,20,22, 24,27,31, 33
NC	Not connected or not used	19,26,29, 30,32,34

† Depending on the kind of printer used, this signal may be called "UNIT SELECT." See your printer manual for more information.

Communications

For hardwiring two 4D's without a modem, use Tandy's RS-232-C cables (Cat. Nos. 26-1490, -1491, -1492, -1493) and null modem adapter (Cat. No. 26-1496).

Appendix C/ Character Codes

Text, control functions, and graphics are represented in the computer by codes. The character codes range from zero through 255.

Code 0 is a prefix code. It tells the video driver to display the special character for codes 1 - 31. These codes are normally treated as cursor control commands.

Codes 1 through 31 normally represent certain control functions. For example, code 13 represents a carriage return or "end of line." These same codes also represent special display characters. To display the special character that corresponds to a particular code (1 - 31), precede the code with a code zero. (Note: Some screen control characters cannot be entered from the TRSDOS Ready or BASIC Ready prompts, but can be directed to the screen by program control. See the CHR$ and ASC functions in the BASIC portion of this manual.)

Codes 32 through 127 represent the text characters -- all those letters, numbers, and other characters that are commonly used to represent textual information.

Codes 128 through 191, when output to the video display, represent 64 graphics characters

Codes 192 through 255, when output to the video display, represent either space compression codes or special or alternate characters, as determined by software. Toggling between these modes is done via codes 21 and 22.

Code 21 toggles the video driver between space compression codes and the special alternate character set. Code 22 toggles the video driver between the special character set and the alternate character set. The setting of the toggle controlled by code 21 determines if the code 22 toggle will have any effect on what is subsequently displayed.

The following chart illustrates the power-up and first toggle states for codes 21 and 22.

	Code 21	**Code 22**
Power-up state	space compression characters	special characters
First toggle state	special alternate characters	alternate characters

At power-up, codes in the range 192 to 255 will produce one or more spaces (space compression mode.) From this point, you can enter the special character set by outputting a code 21 to the display. You can then enter the alternate character set by outputting a code 22 to the display. To switch back to the special set, output another code 22. To switch back to space compression codes from either the special or alternate character set, output a code 21.

When you are in space compression mode, outputting a code 22 still toggles between special and alternate character sets, even though it does not affect the characters subsequently displayed. Any characters in the range 192-255 that are already on the display will toggle between special and alternate character sets each time a code 22 is received.

Note: Special and alternate characters are not available if reverse video (code 16) is enabled.

ASCII Character Set

Code		ASCII		
Dec.	**Hex.**	**Abbrev.**	**Keyboard**	**Video Display**
0	00	NUL	CTRL @	Next character is treated as displayable, if in the range 1 - 31, a special character is displayed (see list of special characters later in this Appendix)
1	01	SOH	CTRL A	
2	02	STX	CTRL B	
3	03	ETX	CTRL C	
4	04	EOT	CTRL D	
5	05	ENQ	CTRL E	
6	06	ACK	CTRL F	
7	07	BEL	CTRL G	
8	08	BS	BACKSPACE or ← or CTRL H	Backspace and erase
9	09	HT	→ or CTRL I	Move cursor to the next tab stop (located every 8 columns)
10	0A	LF	↓ or CTRL J	Move cursor to start of next line
11	0B	VT	↑ or CTRL K	
12	0C	FF	CTRL L	
13	0D	CR	ENTER or CTRL M	Move cursor to start of next line
14	0E	SO	CTRL N	Cursor on
15	0F	SI	CTRL O	Cursor off

Code Dec.	Hex.	ASCII Abbrev.	Keyboard	Video Display
16	10	DLE	CTRL P	Enable reverse video and set high bit routine on*
17	11	DC1	CTRL Q	Set high bit routine off*
18	12	DC2	CTRL R	
19	13	DC3	CTRL S	
20	14	DC4	CTRL T	
21	15	NAK	CTRL U	Swap space compression/ special characters
22	16	SYN	CTRL V	Swap special/alternate characters
23	17	ETB	CTRL W	Set to 40 characters per line
24	18	CAN	SHIFT BACKSPACE or SHIFT ← or CTRL X	Backspace without erasing
25	19	EM	SHIFT → or CTRL Y	Advance cursor
26	1A	SUB	SHIFT ↓ or CTRL Z	Move cursor down
27	1B	ESC	SHIFT ↑ or CTRL ,	Move cursor up
28	1C	FS	CTRL /	Move cursor to upper left corner. Disable reverse video and set high bit routine off.* Set to 80 characters per line.
29	1D	GS	CTRL ENTER or CTRL .	Erase line and start over
30	1E	RS	CTRL ;	Erase to end of line
31	1F	VS	SHIFT CLEAR	Erase to end of display
32	20	SPA	SPACE BAR	(blank)
33	21		!	!
34	22		"	"
35	23		#	#
36	24		$	$

* When the high bit routine is on, characters 20 - 127 are converted to characters 148 - 255. When reverse video is enabled, characters 128 - 191 are displayed as standard ASCII characters in reverse video.

Code Dec.	Hex	Keyboard	Video Display
37	25	%	%
38	26	&	&
39	27	'	'
40	28	(	(
41	29	)	)
42	2A	*	*
43	2B	+	+
44	2C	,	,
45	2D	-	-
46	2E	.	.
47	2F	/	/
48	30	0	0
49	31	1	1
50	32	2	2
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8
57	39	9	9
58	3A	:	:
59	3B	;	;
60	3C	<	<
61	3D	=	=
62	3E	>	>
63	3F	?	?
64	40	@	@
65	41	*A	A

* A - Z (codes 65 - 90) are shifted functions. Hold down SHIFT and then press the desired key. If the CAPS lock is enabled, you need not press SHIFT.

Code Dec.	Hex	Keyboard	Video Display
66	42	B	B
67	43	C	C
68	44	D	D
69	45	E	E
70	46	F	F
71	47	G	G
72	48	H	H
73	49	I	I
74	4A	J	J
75	4B	K	K
76	4C	L	L
77	4D	M	M
78	4E	N	N
79	4F	O	O
80	50	P	P
81	51	Q	Q
82	52	R	R
83	53	S	S
84	54	T	T
85	55	U	U
86	56	V	V
87	57	W	W
88	58	X	X
89	59	Y	Y
90	5A	Z	Z
91	5B	CLEAR ,	[
92	5C	CLEAR /	\
93	5D	CLEAR .	]
94	5E	CLEAR ;	^
95	5F	CLEAR ENTER	_
96	60	SHIFT @	`

Code Dec.	Hex	ASCII Abbrev.	Keyboard	Video Display
97	61		A	a
98	62		B	b
99	63		C	c
100	64		D	d
101	65		E	e
102	66		F	f
103	67		G	g
104	68		H	h
105	69		I	i
106	6A		J	J
107	6B		K	K
108	6C		L	L
109	6D		M	m
110	6E		N	n
111	6F		O	o
112	70		P	p
113	71		Q	q
114	72		R	r
115	73		S	s
116	74		T	t
117	75		U	u
118	76		V	v
119	77		W	w
120	78		X	x
121	79		Y	y
122	7A		Z	z
123	7B		CLEAR SHIFT ,	{
124	7C		CLEAR SHIFT /	|
125	7D		CLEAR SHIFT .	}
126	7E		CLEAR SHIFT ;	˜
127	7F	DEL	CLEAR SHIFT ENTER	±

Extended (non-ASCII) Character Set

Code Dec.	Hex	ASCII Keyboard	Video Display
128	80	BREAK	See Special Character Table
129	81	CLEAR CTRL A or F1	"
130	82	CLEAR CTRL B or F2	"
131	83	CLEAR CTRL C or F3	"
132	84	CLEAR CTRL D	"
133	85	CLEAR CTRL E	"
134	86	CLEAR CTRL F	"
135	87	CLEAR CTRL G	"
136	88	CLEAR CTRL H	"
137	89	CLEAR CTRL I	"
138	8A	CLEAR CTRL J	"
139	8B	CLEAR CTRL K	"
140	8C	CLEAR CTRL L	"
141	8D	CLEAR CTRL M	"
142	8E	CLEAR CTRL N	"
143	8F	CLEAR CTRL O	"
144	90	CLEAR CTRL P	"
145	91	CLEAR CTRL Q or SHIFT F1	"
146	92	CLEAR CTRL R or SHIFT F2	"
147	93	CLEAR CTRL S or SHIFT F3	"
148	94	CLEAR CTRL T	"
149	95	CLEAR CTRL U	"
150	96	CLEAR CTRL V	"
151	97	CLEAR CTRL W	"
152	98	CLEAR CTRL X	"

Code Dec.	Hex	ASCII Keyboard	Video Display
153	99	CLEAR CTRL Y	See Special Character Table
154	9A	CLEAR CTRL Z	"
155	9B	CLEAR SHIFT ↑	"
156	9C		"
157	9D		"
158	9E		"
159	9F		"
160	A0	CLEAR SPACE	"
161	A1	CLEAR SHIFT 1	"
162	A2	CLEAR SHIFT 2	"
163	A3	CLEAR SHIFT 3	"
164	A4	CLEAR SHIFT 4	"
165	A5	CLEAR SHIFT 5	"
166	A6	CLEAR SHIFT 6	"
167	A7	CLEAR SHIFT 7	"
168	A8	CLEAR SHIFT 8	"
169	A9	CLEAR SHIFT 9	"
170	AA	CLEAR SHIFT :	"
171	AB		"
172	AC		"
173	AD	CLEAR -	"
174	AE		"
175	AF		"
176	B0	CLEAR 0	"
177	B1	CLEAR 1	"
178	B2	CLEAR 2	"
179	B3	CLEAR 3	"
180	B4	CLEAR 4	"
181	B5	CLEAR 5	"
182	B6	CLEAR 6	"

Code Dec.	Hex	ASCII Keyboard	Video Display
183	B7	CLEAR 7	See Special Character Table
184	B8	CLEAR 8	"
185	B9	CLEAR 9	"
186	BA	CLEAR :	"
187	BB		"
188	BC		"
189	BD	CLEAR SHIFT -	"
190	BE		"
191	BF		"
192	C0	CLEAR @ *	"
193	C1	CLEAR A **	"
194	C2	CLEAR B **	"
195	C3	CLEAR C **	"
196	C4	CLEAR D **	"
197	C5	CLEAR E **	"
198	C6	CLEAR F **	"
199	C7	CLEAR G **	"
200	C8	CLEAR H **	"
201	C9	CLEAR I **	"
202	CA	CLEAR J **	"
203	CB	CLEAR K **	"
204	CC	CLEAR L **	"
205	CD	CLEAR M **	"
206	CE	CLEAR N **	"
207	CF	CLEAR O **	"
208	D0	CLEAR P **	"
209	D1	CLEAR Q **	"
210	D2	CLEAR R **	"

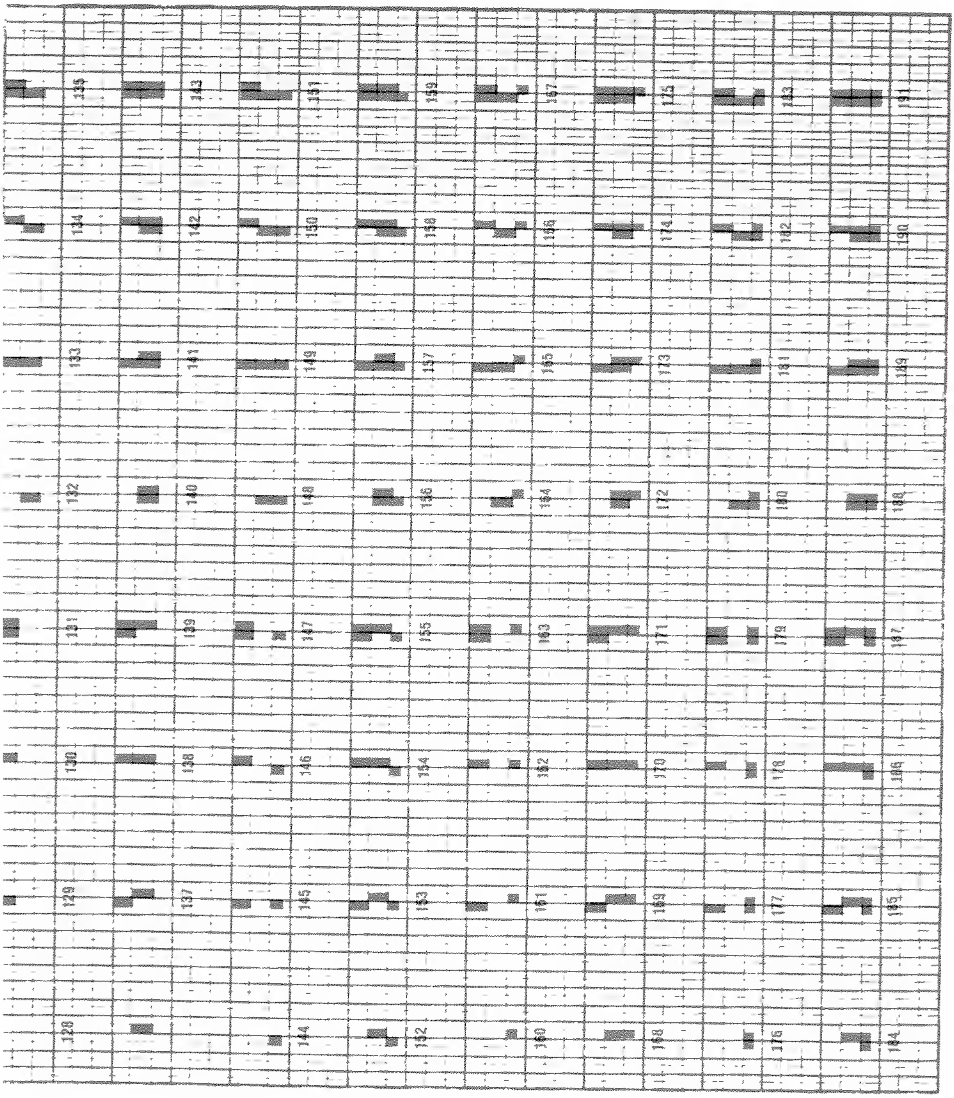
* Empties the type-ahead buffer.
**Used by Keystroke Multiple, if KSM is active.

Code Dec.	Hex	ASCII Keyboard	Video Display
211	D3	CLEAR S **	See Special Character Table
212	D4	CLEAR T **	"
213	D5	CLEAR U **	"
214	D6	CLEAR V **	"
215	D7	CLEAR W **	"
216	D8	CLEAR X **	"
217	D9	CLEAR Y **	"
218	DA	CLEAR Z **	"
219	DB		"
220	DC		"
221	DD		"
222	DE		"
223	DF		"
224	E0	CLEAR SHIFT @	"
225	E1	CLEAR SHIFT A	"
226	E2	CLEAR SHIFT B	"
227	E3	CLEAR SHIFT C	"
228	E4	CLEAR SHIFT D	"

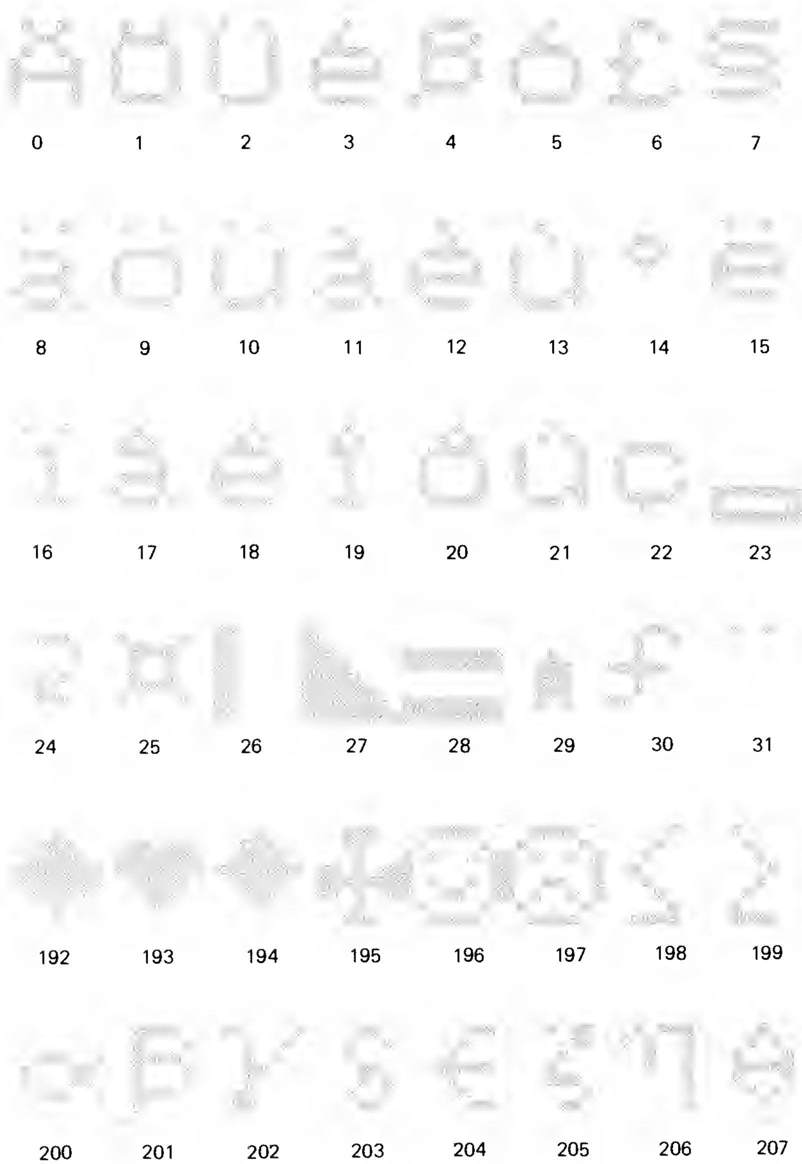
** Used by Keystroke Multiply, if KSM is active.

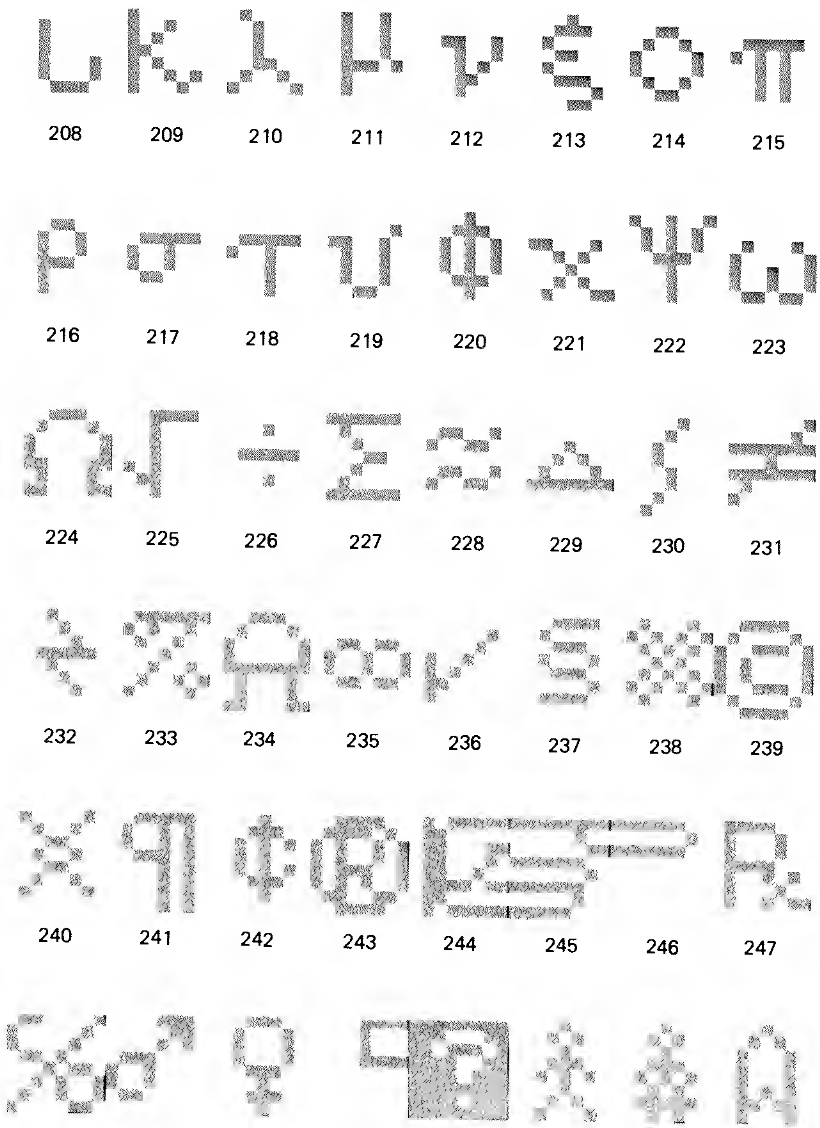
Code Dec.	Hex	ASCII Keyboard	Video Display
229	E5	CLEAR SHIFT E	See Special Character Table
230	E6	CLEAR SHIFT F	"
231	E7	CLEAR SHIFT G	"
232	E8	CLEAR SHIFT H	"
233	E9	CLEAR SHIFT I	"
234	EA	CLEAR SHIFT J	"
235	EB	CLEAR SHIFT K	"
236	EC	CLEAR SHIFT L	"
237	ED	CLEAR SHIFT M	"
238	EE	CLEAR SHIFT N	"
2139	EF	CLEAR SHIFT O	"
240	F0	CLEAR SHIFT P	"
241	F1	CLEAR SHIFT Q	"
242	F2	CLEAR SHIFT R	"
243	F3	CLEAR SHIFT S	"
244	F4	CLEAR SHIFT T	"
245	F5	CLEAR SHIFT U	"
246	F6	CLEAR SHIFT V	"
247	F7	CLEAR SHIFT W	"

Code Dec.	Hex	ASCII Keyboard	Video Display
248	F8	CLEAR SHIFT X	See Special Character Table
249	F9	CLEAR SHIFT Y	"
250	FA	CLEAR SHIFT Z	"
152	FB		"
252	FC		"
253	FD		"
254	FE		"
255	FF		"



Special Characters (0-31, 192-255)




208 209 210 211 212 213 214 215
216 217 218 219 220 221 222 223
224 225 226 227 228 229 230 231
232 233 234 235 236 237 238 239
240 241 242 243 244 245 246 247
248 249 250 251 252 253 254 255

Appendix D/ Error Messages and Problems

In Case Of Difficulty

Your TRSDOS operating system was designed and tested to provide you with trouble-free operation. If you do experience problems, there is a good chance that something other than the TRSDOS system is at fault. This section discusses some of the most common user problems, and suggests general cures for these problems.

Problem 1 . . . The system seems to access the wrong disk drives, or cannot read the diskettes.

If you have trouble reading Model I and III TRSDOS diskettes, refer to the REPAIR and CONV Utilities. Those sections explain how to make these types of disks readable.

If your system seems to access the wrong disk, reset your computer. You may have selected some combination of options that are preventing the system from functioning properly.

Remember that when you specify a drive number, you are specifying a logical drive number which, based on your system's configuration, may point at physical drives in another order. If you have SYSGENed these settings, you may have to hold (CLEAR) down while you reset your computer. This insures that all drives are in their default state.

Problem 2 . . . RS-232-C communications do not work, or function incorrectly.

If you experience RS-232-C problems, the first thing you should do is to make sure both "ends" are operating with the same RS-232-C parameters (baud rate, word length, stop bits, and parity). If these parameters are not the same at each end, the data sent and received appears scrambled.

Some hardware, such as serial printers, require "handshaking" when running above a certain baud rate. It may be necessary to hook the hardware's handshake line (such as the BUSY line) to an appropriate RS-232-C lead, such as CTS.

Problem 3 . . . Random system crashes, recurring disk I/O errors, system lock-up, and other random glitches keep happening.

If you encounter these types of problems, the first thing to check is the cable connections between the TRS-80 and the peripherals.

If you experience constant difficulty in disk read/write operations, it is possible that the disk drive heads need cleaning. There are kits available at your Radio Shack store to clean disk heads, or you may wish to have the disk drive serviced at a repair facility. If you need to frequently clean the disk heads, you might be using some defective disk media. Check the diskettes for any obvious signs of flaking or excess wear, and dispose of any that appear

even marginal. Tobacco smoke and other airborne contaminants can build up on disk heads, and can cause read/write problems. Disk drives in "dirty" locations may need to have their heads cleaned as often as once a week.

One common and often overlooked cause of random-type problems is static electricity. In areas of low humidity, static electricity is present, even if actual static discharges are not felt by the computer operator. Be aware that static discharges can cause system glitches, as well as physically damage computer hardware and disk media.

Error Messages

If the computer displays one of the messages listed in this appendix, an operating system error occurred. Any other error message refers to an application program error, and you should see your application program manual for an explanation.

When an error message is displayed:

- Look up operating system errors below and take any recommended actions. (See your application program manual for explanations of application program errors.)
- Try using other diskettes.
- Reset the computer and try the operation again.
- Check all the power connections.
- Check all interconnections.
- Remove all diskettes from drives, turn off the computer, wait 15 seconds, and turn it on again.
- If you try all these remedies and still get an error message, contact a Radio Shack Service Center.

NOTE: If there is more than one thing wrong, the computer might wait until you correct the first error before displaying the second error message.

This list of error messages is alphabetical, with the decimal and hexadecimal error numbers in parentheses. Following it is a quick reference list of the messages arranged in numerical order.

TRSDOS Error Messages

Attempted to read locked/deleted data record **(Error 7, X'07')**

In a system that supports a "deleted record" data address mark, an attempt was made to read a deleted sector. TRSDOS currently does not use the deleted sector data address mark. Check for an error in your application program.

Attempted to read system data record **(Error 6, X'06')**

An attempt was made to read a directory cylinder sector without using the directory read routines. Directory cylinder sectors are written with a data address mark that differs from the data sector's data address mark. Check for an error in your application program.

Data record not found during read **(Error 5, X'05')**

The sector number for the read operation is not on the cylinder being referenced. Either the disk is flawed, you requested an incorrect number, or the cylinder is improperly formatted. Try the operation again. If it fails, use a different disk. Reformatting the old disk should lock out the flaw.

Data record not found during write **(Error 13, X'0D')**

The sector number requested for the write operation cannot be found on the cylinder being referenced. Either the disk is flawed, you requested an incorrect number, or the cylinder is improperly formatted. Try the operation again. If it fails use another disk.

Device in use **(Error 39, X'27')**

A request was made to REMOVE a device (delete it from the Device Control Block tables) while it was in use. RESET the device in use before removing it.

Device not available **(Error 8, X'08')**

A reference was made for a logical device that cannot be found in the Device Control Block. Probably, your device specification was wrong or the device peripheral was not ready. Use the DEVICE command with the (B) parameter to display all devices available to the system.

Directory full -- can't extend file **(Error 30, X'1E')**

A file has all extent fields of its last directory record in use and must find a spare directory slot but none is available. (See the "Directory Records" section.) Copy the disk's files to a newly formatted diskette to reduce file fragmentation. You may use backup by class or backup reconstruct to reduce fragmentation.

Directory read error **(Error 17, X'11')**

A disk error occurred during a directory read. The problem may be media, hardware, or program failure. Move the disk to another drive and try the operation again.

Directory write error **(Error 18, X'12')**

A disk error occurred during a directory write to disk. The directory may no longer be reliable. If the problem recurs, use a different diskette.

Disk space full **(Error 27, X'1B')**

While a file was being written, all available disk space was used. The disk contains only a partial copy of the file. Write the file to a diskette that has more available space. Then, REMOVE the partial copy to recover disk space.

End of file encountered **(Error 28, X'1C')**

You tried to read past the end of file pointer. Use the DIR command to check the size of the file. This error also occurs when you use the @PEOF supervisor call to successfully position to the end of a file. Check for an error in your application program.

Extended error **(Error 63)**

An error has occurred and the extended error code is in the HL register pair.

File access denied **(Error 25, X'19')**

You specified a password for a file that is not password protected or you specified the wrong password for a file that is password protected.

File already open **(Error 41, X'29')**

You tried to open a file for UPDATE level or higher, and the file already is open with this access level or higher. This forces a change to READ access protection. Use the RESET library command to close the file.

File not in directory **(Error 24, X'18')**

Check the spelling of the filespec. Use the DIR command to see if the file is on the disk.

File not open **(Error 38, X'26')**

You requested an I/O operation on an unopened file. Open the file before access.

GAT read error (Error 20, X'14')

A disk error occurred during the reading of the Granule Allocation Table. The problem may be media, hardware, or program failure. Move the diskette to another drive and try the operation again.

GAT write error (Error 21, X'15')

A disk error occurred during the writing of the Granule Allocation Table. The GAT may no longer be reliable. If the problem recurs, use a different drive or different diskette.

HIT read error (Error 22, X'16')

A disk error occurred during the reading of the Hash Index Table. The problem may be media, hardware, or program failure. Move the diskette to another drive and try the operation again.

HIT write error (Error 23, X'17')

The Hash Index Table may no longer be reliable. If the problem recurs, try the operation again, using a different drive. If it still fails, use a different disk.

Illegal access attempted to protected file (Error 37, X'25')

The USER password was given for access to a file, but the requested access required the OWNER password. (See the ATTRIB command.) Another cause is an attempt to write to a write-protected disk or an already-open file.

Illegal drive number (Error 32, X'20')

The specified disk drive is not included in your system or is not ready for access (no diskette, non-TRSDOS Version 6 diskette, drive door open, and so on). See the DEVICE command.

Illegal file name (Error 19, X'13')

The specified filespec does not meet TRSDOS filespec requirements. See Chapter 1 for proper filespec syntax.

Illegal logical file number (Error 16, X'10')

Your program probably has altered the File Control Block improperly. Check for an error in your application program.

Load file format error (Error 34, X'22')

An attempt was made to load a file that cannot be loaded by TRSDOS. The file was probably a data file or a BASIC program file.

Lost data during read (Error 3, X'03')

Information was not transferred in the time allotted; therefore, it was lost. Try the operation again, using a different drive. If it still fails, use a different disk.

Lost data during write (Error 11, X'0B')

Information was not transferred in the time allotted; therefore, it was lost. Try the operation again, using a different drive. If it still fails, use a different disk.

LRL open fault (Error 42, X'2A')

The logical record length specified when the file was opened is different than the LRL used when the file was created. COPY the file to another file that has the specified LRL.

No device space available (Error 33, X'21')

You tried to SET a driver or filter and all of the Device Control Blocks were in use. Use the DEVICE command to see if any non-system devices can be removed to provide more space.

No directory space available (Error 26, X'1A')

You tried to open a new file and no space was left in the directory. Use a different disk or REMOVE some files you no longer need.

No error (Error 0)

The @ERROR supervisor call was called without any error condition being detected. A return code of zero indicates no error. Check for an error in your application program.

Parameter error (Error 44 X'2C')

An error occurred while executing a command line or utility because a parameter that does not exist was specified. Check the spelling of the parameter name, value, or abbreviation.

Parity error during header read (Error 1, X'01')

During a sector I/O request, the system could not read the sector header successfully. If this error occurs repeatedly, the problem is probably media or hardware failure. Try the operation again, using a different drive or diskette.

Parity error during header write (Error 9, X'09')

During a sector write, the system could not write the sector header satisfactorily. If this error occurs repeatedly, the problem is probably media or hardware failure. Try the operation again, using a different drive or diskette.

Parity error during read (Error 4, X'04')

An error occurred during a sector read. Its probable cause is media failure or a dirty or faulty disk drive. Try the operation again, using a different drive or diskette.

Parity error during write (Error 12, X'0C')

An error occurred during a sector write operation. Its probable cause is media failure or a dirty or faulty disk drive. Try the operation again, using a different drive or diskette.

Program not found (Error 31, X'1F')

The file cannot be loaded because it is not in the directory. Either the filespec was misspelled or the disk that contains the file was not loaded.

Protected system device (Error 40, X'28')

You cannot REMOVE any of the following devices: *KI, *DO, *PR, *JL, *SI, *SO.

Record number out of range (Error 29, X'1D')

A request to read a record within a random access file provided a record number that was beyond the end of the file. Correct the record number or try again using another copy of the file.

Seek error during read (Error 2, X'02')

During a read sector disk I/O request, the cylinder that should contain the sector was not found within the time allotted. (The time is set by the step rate specified in the Drive Code Table.) Either the cylinder is not formatted or it is no longer readable or the step rate is too low for the hardware to respond. You can set an appropriate step rate using the SYSTEM library command. The problem may also be caused by media or hardware failure. In this case, try the operation again, using a different drive or diskette.

Seek error during write (Error 10, X'0A')

During a sector write. the cylinder that should contain the sector was not found within the time allotted. (The time is set by the step rate specified in the Drive Code Table.) Either the cylinder is not formatted or it is no longer readable, or the step rate is too low for the hardware to respond. You can set an appropriate step rate using the SYSTEM library command. The problem may also be caused by media or hardware failure. In this case, try the operation again, using a different drive or diskette.

— Unknown error code

The @ ERROR supervisor call was called with an error number that is not defined. Check for an error in your application program.

Write fault on disk drive **(Error 14, X'0E')**

An error occurred during a write operation. This probably indicates a hardware problem. Try a different diskette or drive. If the problem continues, contact a Radio Shack Service Center.

Write protected disk **(Error 15, X'0F')**

Remove the write-protect tab, if the diskette has one. If it does not, use the DEVICE command to see if the drive is set as write protected. If it is, you can use the SYSTEM command with the (WP=OFF) parameter to write enable the drive. If the problem recurs, check the drive connections on the external drives, even if the error is occurring on an internal drive. Or, use a different drive or diskette.

TRSDOS ERROR MESSAGES

Decimal	Hex	Message
0	X'00'	No Error
1	X'01'	Parity error during header read
2	X'02'	Seek error during read
3	X'03'	Lost data during read
4	X'04'	Parity error during read
5	X'05'	Data record not found during read
6	X'06'	Attempted to read system data record
7	X'07'	Attempted to read locked deleted data record
8	X'08'	Device not available
9	X'09'	Parity error during header write
10	X'0A'	Seek error during write
11	X'0B'	Lost data during write
12	X'0C'	Parity error during write
13	X'0D'	Data record not found during write
14	X'0E'	Write fault on disk drive
15	X'0F'	Write protected disk
16	X'10'	Illegal logical file number
17	X'11'	Directory read error
18	X'12'	Directory write error
19	X'13'	Illegal file name
20	X'14'	GAT read error
21	X'15'	GAT write error
22	X'16'	HIT read error
23	X'17'	HIT write error
24	X'18'	File not in directory
25	X'19'	File access denied
26	X'1A'	Full or write protected disk
27	X'1B'	Disk space full
28	X'1C'	End of file encountered
29	X'1D'	Record number out of range
30	X'1E'	Directory full — can't extend file
31	X'1F'	Program not found
32	X'20'	Illegal drive number
33	X'21'	No device space available
34	X'22'	Load file format error
37	X'25'	Illegal access attempted to protected file
38	X'26'	File not open
39	X'27'	Device in use
40	X'28'	Protected system device
41	X'29'	File already open
42	X'2A'	LRL open fault
43	X'2B'	SVC parameter error
44	X'2C'	Parameter error
63	X'3F'	Extended error
—		Unknown error code

BASIC Error Codes and Messages

Number	Message
1	NEXT without FOR
	A variable in a NEXT statement does not correspond to any previously executed FOR statement variable.
2	Syntax error
	BASIC encountered a line that contains an incorrect sequence of characters (such as unmatched parenthesis, misspelled statement, incorrect punctuation, etc.). BASIC automatically enters the edit mode at the line that caused the error.
3	RETURN without GOSUB
	BASIC encountered a RETURN statement for which there is no matching GOSUB statement.
4	Out of DATA
	BASIC encountered a READ statement, but no DATA statements with unread items remain in the program.
5	Illegal function call
	A parameter that is out of range was passed to a math or string function. An FC error may also occur as the result of:

a. A negative or unreasonably large subscript.

b. A negative or zero argument with LOG.

c. A negative argument to SQR.

d. A negative mantissa with a noninteger exponent.

e. A call to a USR function for which the starting address has not yet been given.

f. An improper argument to MID$, LEFT$, RIGHT$, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON . . . GOTO.

6 Overflow

The result of a calculation was too large to be represented in BASIC numeric format. If underflow occurs, the result is zero and execution continues without an error.

7 Out of memory

A program is too large, or has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.

8 Undefined line number

A nonexistent line was referenced in a GOTO, GOSUB, IF . . . THEN . . . ELSE, or DELETE statement.

9 Subscript out of range

An array element was referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.

10 Duplicate Definition

Two DIM statements were given for the same array, or a DIM statement was given for an array after the default dimension of 10 has been established for that array.

11 Division by zero

An expression includes division by zero, or the operation of involution results in zero being raised to a negative power. BASIC supplies machine infinity with the sign of the numerator as the result of the division, or it supplies positive machine infinity as the result of the involution. Execution then continues.

12 Illegal direct

A statement that is illegal in direct mode was entered as a direct mode command.

13 Type mismatch

A string variable name was assigned a numeric value or vice versa. A numeric function was given a string argument or vice versa.

14 Out of string space

String variables have caused BASIC to exceed the amount of free memory remaining. BASIC allocates string space dynamically, until it runs out of memory.

15 String too long

An attempt was made to create a string more than 255 characters long.

16 String formula too complex

A string expression is too long or too complex. The expression should be broken into smaller expressions.

17 Can't continue

An attempt was made to continue a program that:

a. Has halted due to an error.

b. Has been modified during a break in execution.

c. Does not exist.

18 Undefined user function

A USR function was called before providing a function definition (DEF statement).

19 No RESUME

An error-handling routine was entered without a matching RESUME statement.

20 RESUME without error

A RESUME statement was encountered prior to an error-handling routine.

21 Unprintable error

An error message is not available for the error that occurred.

22 Missing operand

An expression contains an operator with no operand.

23 Line buffer overflow

An attempt was made to input a line with too many characters.

26 FOR without NEXT

A FOR statement was encountered without a matching NEXT.

29 WHILE without WEND

A WHILE statement does not have a matching WEND.

30 WEND without WHILE

A WEND statement was encountered without a matching WHILE.

Disk Errors

50 FIELD overflow

A FIELD statement is attempting to allocate more bytes than were specified for the record length of a direct-access file.

51 Internal error

An internal malfunction has occurred in BASIC. Report to Radio Shack the conditions under which the message appeared.

52 Bad file number

A statement or command references a file with a buffer number that is not OPEN or is out of the range of file numbers specified at initialization.

53 File not found

A LOAD, KILL, or OPEN statement references a file that does not exist on the current disk.

54 Bad file mode

An attempt was made to use PUT, GET, or LOF with a sequential file, to LOAD a direct file, or to execute an OPEN statement with a file mode other than I, O, R, E or D.

55 File already open

An OPEN statement for sequential output was issued for a file that is already open; or a KILL statement was given for a file that is open.

57 Device I/O error

An Input/Output error occurred. This is a fatal error; the operating system cannot recover from it.

58 File already exists

The filespec specified in a NAME statement is identical to a filespec already in use on the disk.

61 Disk full

All disk storage space is in use.

62 Input past end

An INPUT statement was executed after all the data in the file had been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end-of-file.

63 Bad record number

In a PUT or GET statement, the record number is either greater than the maximum allowed (65,535) or equal to zero.

64 Bad file name

An illegal filespec (file name) was used with a LOAD, SAVE, KILL, or OPEN statement (for example, a filespec with too many characters).

66 Direct statement in file

A direct statement was encountered while LOADing an ASCII-format file. The LOAD is terminated.

67 Too many files

An attempt was made to create a new file (using SAVE or OPEN) when all directory entries are full.

68 Disk write protected

The drive number you specified is write-protected. Either the write-protect notch is covered on the diskette or the diskette was write-protected with the TRSDOS SYSTEM (DRIVE=n,WP) command.

69 File access DENIED

The file you attempted to access is password protected. You must specify the exact password to gain access to the file.

70 Command Aborted

The command specified in a SYSTEM statement was aborted. You either pressed (BREAK) during the execution of a library command or tried to execute a program or utility. Only library commands can be executed with the SYSTEM statement.

Appendix E/ Converting Model III BASIC Programs to Model 4 Mode

You can run a Model III Basic applications program on your computer. However, you may need to make a few changes to the program. The differences between Model III and Model 4 BASIC are listed below.

1. **ROM Subroutines**. Model III BASIC is a ROM- and RAM-based language. Model 4 BASIC is strictly a RAM language; therefore, it cannot access any of the Model III's ROM subroutines.
2. **Disk Files**. Model 4 BASIC does not provide cassette support. It is exclusively a "disk system", that is, you can only use it with floppy diskettes or with a hard disk system. If you have learned BASIC through *Getting Started with TRS-80 BASIC*, or have never worked with a disk system before, read about "Disk Files" in Chapter 5. This chapter explains how you can store and access data on disk. You also need to read Chapter 1, "Sample Session", which describes how to load disk BASIC and how to save a program on disk.
3. **Characters per Line**. Both BASICs allow you to type up to 255 characters per line. However, there is a slight difference. With Model 4 BASIC, you can type up to 249 characters per line. The other six characters are reserved for the line number and the space following the line number. With Model III BASIC, you can type up to 240 characters in the command mode, and add the extra 15 characters in the edit mode.
4. **Variable Names**. Model III BASIC recognizes only the first two letters of a variable name; Model 4 BASIC allows variable names of up to 40 characters, all of which are significant.
5. **Converting to Integers**. In converting a single- or double-precision number to integer value, Model III BASIC truncates the number; Model 4 BASIC rounds the number. This difference in conversions also affects assignment statements and function or statement evaluations. For example, if you type I% = 2.5, Model III BASIC converts 2.5 to 2; Model 4 BASIC converts it to 3. If you type TAB(4.5), Model III BASIC moves to the fourth tab position; Model 4 BASIC moves to the fifth tab position.

   If you enter a number as a constant in response to a command that calls for an integer, and the number is out of integer range, BASIC converts it to single or double precision. When the number is printed, it appears with a type-declaration tag at the end.
6. **Print Width**. In Model III mode, the screen displays up to 64 characters horizontally. In Model 4 mode, it displays up to 80 characters horizontally.

7. **Memory Addresses and I/O Ports**. ROM and RAM locations that are used in some Model III BASIC programs are unavailable in Model 4 mode. See the *Model 4/4P Technical Reference Manual* for information on the location of ROM routines. (There are no ROM routines in Model 4 mode.)

   Additional I/O ports are available in the Model 4 hardware. Some of the Model III ports perform additional functions, as well. See the *Model 4/4P Technical Reference Manual*.

8. **BASIC Keywords**. The following Model III BASIC keywords are not supported by Model 4 BASIC: CSAVE, CLOAD, POINT, CLOCK, CMD, POSN, RENAME, and VERIFY. In Model 4 BASIC, the SYSTEM statement performs some of the same functions as the CMD statement in MODEL III BASIC.

9. **Reserved Words**. Model 4 BASIC **requires that all reserved words be delimited by spaces.** Only those characters which may be part of the keyword's syntax can be typed immediately after or before the keyword. For all other characters, leave a space between the keyword and the character. (For example, you cannot type DEFUSR; you must leave a space between DEF and USR.) Appendix F includes a listing of Reserved Words.

10. **Error Messages**. Model 4 BASIC's error codes, character codes, and internal codes for BASIC keywords are different than Model III BASIC's codes. See the appendices for more information.

11. **String Space**. Model 4 BASIC allocates string space dynamically; you do not need to allocate string space with the CLEAR statement. Instead, use CLEAR to set the maximum memory location BASIC may access and the amount of stack space. For more information, see CLEAR in Chapter 7.

12. **Printing Single and Double-Precision Numbers**. The rules for printing single and double-precision numbers are different. For more information, see PRINT in Chapter 7.

13. **Division by Zero**. Contrary to Model III BASIC, Model 4 BASIC does not produce a fatal error if it encounters division by zero or overflow. Instead, it prints an error message and continues executing your program.

14. **FOR . . . NEXT**. Model 4 BASIC skips the body of a FOR . . . NEXT loop if the initial value of the loop, times the sign of the STEP, exceeds the final value of the loop, times the sign of the STEP. For a more detailed explanation, see FOR . . . NEXT in Chapter 7.

15. **Nested Subroutines.** If your program has deeply nested subroutines or nested FOR . . . NEXT loops, an "Out of memory" error may occur. To avoid this, use the CLEAR statement to set aside additional stack space for your subroutines. See CLEAR in Chapter 7 for more information. It is also illegal to exit a FOR . . . NEXT loop without terminating the loop.

16. **IF . . . THEN . . . or IF . . . . THEN . . . ELSE.** With Model III BASIC, the word "THEN" is optional in both of these statements. With Model 4 BASIC, it is required.

17. **PRINT@ and PRINT TAB.** If a string is too long to fit on the current line, Model 4 BASIC prints the entire string on the next line. Model III BASIC prints as many characters as possible on the first line, and the rest on the second line.

    In a PRINT TAB(n) statement, if *n* is greater than 80, Model 4 BASIC divides *n* by 80. The remainder of this division is used as the tab position. For example, if you type TAB(91), Model 4 BASIC tabs to position 11 on the screen. Model III BASIC tabs to position 91.

18. **Self-Documenting Programs.** Model 4 BASIC programs can be self-documenting, as in the following example.

```
100 INPUT EFFORT
110 INPUT DISTANCE
120 FORCE = EFFORT * DISTANCE
130 PRINT FORCE
140 END
```

    Under Model 4 BASIC, the reserved words (FOR and TAN in the above example) in the variable names will not cause a syntax error. This is because they must be delimited by surrounding spaces, in order to be recognized as reserved words.

    Model III BASIC would return syntax errors in this example.

19. **Graphics Characters.** Under Model 4 BASIC, the size of the graphics characters is different than under Model III BASIC. The lowest portions of the Model 4 BASIC graphics characters are smaller than their Model III BASIC equivalents.

20. **Input.** In Model III BASIC, if you do not supply a number or a string for an INPUT statement (you simply press (ENTER)), the variables in the statement retain their previous values.

    If you do the same in Model 4 BASIC, numeric variables are set to 0 and strings variables are set to a length of 0 (null string).

21. **Width.** In Model III BASIC, if a string is longer than the width of the screen, BASIC breaks it at the 64th column and continues it on the next line.

Model 4 BASIC, on the other hand, uses a *sizing* function to keep the string on one line whenever possible. To determine if it **is** possible, BASIC adds the current position of the cursor to the length of the string to be printed. If the sum is greater than the width of the screen, BASIC displays the string at the beginning of the next line. If the cursor was already at the beginning of a line, BASIC must break the string.

To disable the sizing function so that your BASIC behaves like Model III BASIC in this regard, use WIDTH to set the screen width to 255. (The width at startup is 80.)

Appendix F/ BASIC Keywords and Derived Functions

Reserved BASIC Keywords

Keyword	Internal Code	Keyword	Internal Code
ABS	65414	FRE	65423
AND	248	GET	193
ASC	65429	GOSUB	141
ATN	65422	GOTO	137
AUTO	171	HEX$	65434
CALL	182	IF	139
CDBL	65438	IMP	252
CHAIN	185	INKEY$	224
CHR$	65430	INP	65424
CINT	65436	INPUT	133
CLEAR	146	INSTR	219
CLOSE	195	INT	65413
CLS	159	KILL	200
COMMON	184	LEFT$	65409
CONT	153	LEN	65426
COS	65420	LET	136
CSNG	65437	LINE	177
CVD	65452	LIST	147
CVI	65450	LLIST	158
CVS	65451	LOAD	196
DATA	132	LOC	65454
DATE$	222	LOF	65455
DEF	151	LOG	65418
DEFDBL	176	LPOS	65435
DEFINT	174	LPRINT	157
DEFSNG	175	LSET	201
DEFSTR	173	MEM	225
DELETE	170	MERGE	197
DIM	134	MID$	65411
EDIT	167	MKD$	65458
ELSE	162	MKI$	65456
END	129	MKS$	65457
EOF	65453	MOD	253
EQV	251	NAME	199
ERASE	166	NEW	148
ERL	215	NEXT	131
ERR	216	NOT	214
ERROR	168	OCT$	65433
ERRS$	223	ON	149
EXP	65419	OPEN	191
FIELD	192	OPTION	186
FIX	65439	OR	249
FN	212	OUT	156
FOR	130	PEEK	65431

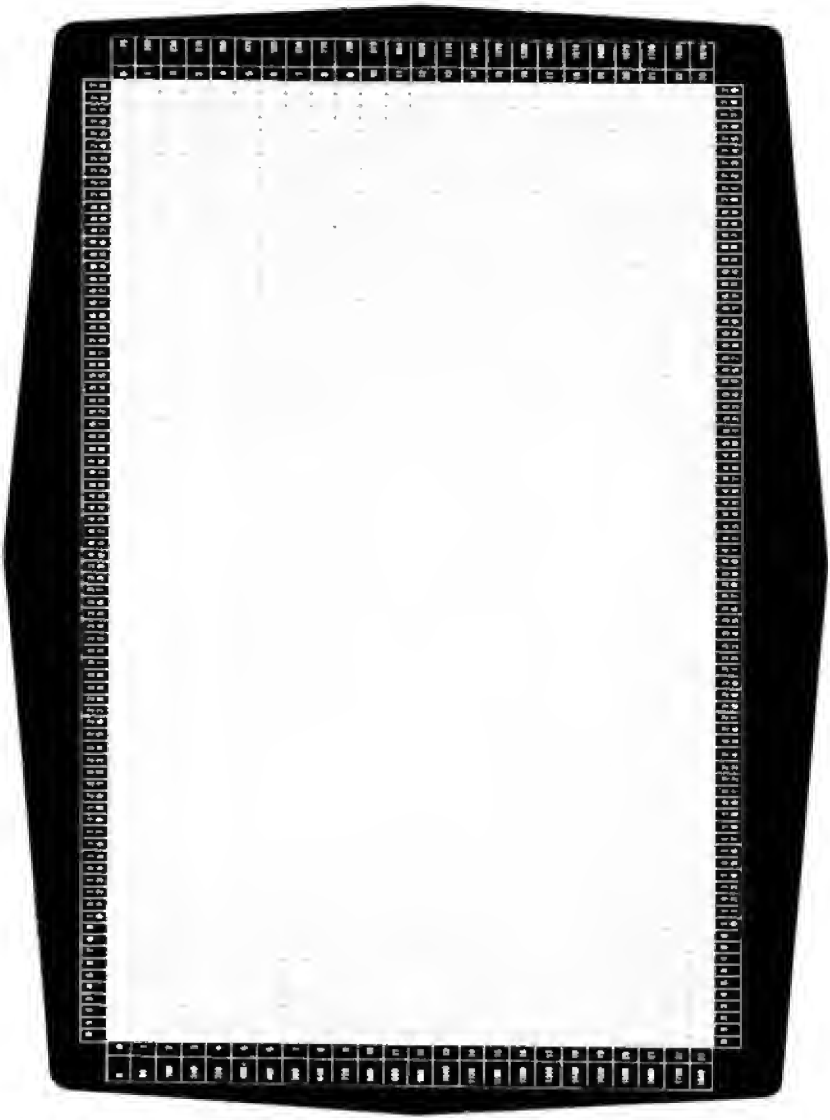
Keyword	Internal Code	Keyword	Internal Code
POKE	152	SYSTEM	189
POS	65425	TAB	209
PRINT	145	TAN	65421
PUT	194	THEN	208
RANDOM	187	TIME$	226
READ	135	TO	207
REM	143	TROFF	164
RENUM	172	TRON	163
RESTORE	140	USING	218
RESUME	169	USR	211
RETURN	142	VAL	65428
RIGHT$	65410	VARPTR	221
RND	65416	WAIT	150
ROW	65459	WEND	181
RSET	202	WHILE	180
RUN	138	WIDTH	161
SAVE	203	WRITE	183
SGN	65412	XOR	250
SIN	65417	+	243
SOUND	205	–	244
SPACE$	65432	*	245
SPC	213	/	246
SQR	65415	^	247
STEP	210	\	254
STOP	144	,	220
STR$	65427	>	240
STRING$	217	=	241
SWAP	165	<	242

Derived BASIC Functions

Functions which are not intrinsic to BASIC may be calculated as follows:

Function	BASIC Equivalent
SECANT	SEC(X) = 1/COS(X)
COSECANT	CSC(X) = 1/SIN(X)
COTANGENT	COT(X) = 1/TAN(X)
INVERSE SINE	ARCSIN(X) = ATN(X/SQR(− X*X + 1))
INVERSE COSINE	ARCCOS(X) = − ATN(X/SQR(− X*X + 1)) + 1.5708
INVERSE SECANT	ARCSEC(X) = ATN(X/SQR(X*X − 1)) + (SGN(X) − 1)*1.5708
INVERSE COSECANT	ARCCSC(X) = ATN(X/SQR(X* X − 1)) + (SGN(X) − 1)*1.5708
INVERSE COTANGENT	ARCCOT(X) = ATN(X) + 1.5708
HYPERBOLIC SINE	SINH(X) = (EXP(X) − EXP(− X))/2
HYPERBOLIC COSINE	COSH(X) = (EXP(X) + EXP(− X))/2
HYPERBOLIC TANGENT	TANH(X) = (EXP(X) − EXP(− X))/(EXP(X) + EXP(− X))
HYPERBOLIC SECANT	SECH(X) = 2/(EXP(X) + EXP(− X))
HYPERBOLIC COSECANT	CSCH(X) = 2/(EXP(X) − EXP(− X))
HYPERBOLIC COTANGENT	COTH(X) = (EXP(X) + (EXP(− X))/(EXP(X) − EXP(− X))
INVERSE HYPERBOLIC SINE	ARCSINH(X) = LOG(X + SQR(X*X + 1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X) = LOG(X + SQR(X*X − 1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X) = LOG((1 + X)/(1 − X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X) = LOG((SQR(− X*X + 1) + 1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X) = LOG((SGN(X)*SQR(X* X + 1) + 1)/X)
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X) = LOG((X + 1)/(X − 1))/2

Appendix G/ Video Display Worksheet



Appendix H/ Glossary

alphanumeric — consisting of only the letters A-Z, a-z, and the numerals 0-9.

ASCII — American Standard Code for Information Interchange, defines seven bit combinations that represent letters, numbers, punctuation, and control codes.

ASCII files — Files that are readable by LISTing the file. Source, text. and data files are usually ASCII files.

background task — A job performed by the computer that is not apparent to the user or does not require interaction with the user. Some examples are the real time CLOCK, the SPOOLer, and the TRACE function.

baud — Refers to the rate of serial data transfer.

bit — One eighth of a byte; one binary digit.

boot — The process of resetting your computer and loading in the resident operating system from the system drive.

buffer — An area in RAM that temporarily holds information that is being passed between devices or programs.

byte — The unit that represents one character to the Model 4. It is composed of eight binary "bits" that are either ON (1) or OFF (0). One byte can represent a number from 0 to 255.

COMM — A communications program capable of interacting with: disk, printer, video display, keyboard, and the RS232 interface. COMM dynamically buffers all of the system devices.

concatenate — To add one variable or string onto the end of another.

configuration — The status of the system and physical devices that are available to it. This configuration can be dynamically changed with several library commands, and can be saved with the SYSGEN library command. If the system is SYSGENed, the SYSGENed configuration is re-established each time the machine is reset or re-started.

cursor — The location on the video display where the next character is printed. It is marked by the presence of a cursor character.

cylinder — All tracks of the same number on a disk drive. On single sided drives, cylinders are the same as tracks.

DAM (Data Address Mark) — A control byte that prefixes each sector on a disk. This byte indicates the type of sector that is about to be read. It can mark a sector as being deleted or undeleted, a user sector or a system sector.

DCB — Device Control Block, a small piece of memory used to control the status. input, and output of data between the system and the devices.

DCT — Drive Code Table, a table in the operating system that contains information about the type of the drive, the number of cylinders, the number of granules per track, how to access the hardware and other control information. The DCT tells the operating system how the logical drives interface with the physical drives.

density — Refers to the density of the data written to a diskette. Double density provides approximately 80% more capacity than single density.

device — The two types of devices are Logical and Physical.

A logical device is one that is referred to in TRSDOS. Logical devices have devspecs, a 2-character name that is prefixed with an asterisk (*). An example of a logical device is *PR, which is normally used to send data to the printer.

A physical device is a piece of hardware, such as the video display or printer. A piece of software called a "driver" connects the logical device to the physical device by translating data from the format used by logical devices into the format required by the hardware, and vice versa.

devspec — The name associated with a device by which it is referenced. A devspec always consists of three characters: an asterisk followed by two alphabetic characters.

directory — An area of a disk that contains the names of the files on the disk, information on where the data in those files is stored on the disk, and other information such as any password, the logical record length, the modification date, and so on.

disk I.D. — A disk's name, master password, and creation date that TRSDOS assigns when the disk is formatted or backed up.

disk name — The name assigned to a disk when it is formatted. On a hard disk each logical drive has its own disk name.

***DO** — The Video Display device.

:drive — Indicate that a drive number can be inserted where this is used. A drive number must always be preceded immediately by a ":".

driver — A program that interfaces a physical device (a piece of hardware) to a logical device, which can be referenced by TRSDOS. Some application programs contain their own driver programs.

EOF — End of File, a marker used to denote the end of a program or data file.

/ext — The extension of a filespec. The use of /ext is sometimes optional. An extension's first character must be a "/" (slash) which is followed by one to three alphanumeric characters, the first of which must be a letter.

FCB — File Control Block, a small piece of memory used to control the status and I/O of data between the operating system and disk files.

filename — The mandatory name used to reference a disk file. A filename consists of one to eight alphanumeric characters, the first of which must be alphabetic.

filespec — A disk file's name. A filespec consists of four fields and two switches. The first field is always mandatory. A filespec is in the following format:

!filename/ext.password:drive!

"!" — (preceding filename) is an optional switch. If you specify this switch, you can build a file with the same name as a TRSDOS command or utility. For example, you can issue the command: LIST DEVICE and TRSDOS will list the user-created file named DEVICE.

filename — The mandatory name of the file.

/ext — The optional file extension.

.password — The optional file password.

:drive — The optional drive number.

"!" — (following :d) is an optional switch. If this switch is set, the end of file marker for filespec is updated after every write to the file.

filter — A machine language program that monitors and/or alters I/O that passes through it. FILTER is also the library command that establishes a FILTER routine.

/FIX — The desired file extension for a PATCH file.

foreground task — Jobs performed by the computer that are apparent to the user, such as running an applications program.

gran — The abbreviation of granule. A gran is the minimum amount of storage used for a disk file. As files are extended, file allocation is increased in increments of grans. The size of a gran varies with the size and density of a diskette.

Hexadecimal — A Base 16 numbering system used in binary computers. Valid hexadecimal digits are 0 through 9 and the letters A through F.

HIGH$ — The name of a memory location in the operating system that contains the address of the highest unprotected memory

address available for use. Programs that are above this location are protected from other programs. You can display or change the value of HIGH$ by using the MEMORY command or the @HIGH$ SVC.

interrupt — A signal generated by the hardware which causes the system to stop what it is doing to perform some other service. These interruptions are used to perform background tasks such as checking the keyboard for input and supplying data to the printer if the spooler is running. After the interrupt is handled the CPU returns to what it was doing before the interrupt.

I/O — The abbreviation for Input/Output.

/JCL — The desired file extension for a DO file. JCL is the abbreviation for Job Control Language.

***JL** — The Joblog device.

***KI** — The Keyboard device.

/KSM — The desired file extension for a ksm file. KSM is an abbreviation for Key-Stroke Multiply.

library — A set of commonly used commands that are grouped together in one file. TRSDOS contains three system libraries.

load module format — A file format that loads directly to a specified RAM address.

LSB — The Least Significant Byte. In memory, the LSB is stored before the MSB. In a register, the LSB is stored after the MSB.

macro — a group of instructions that can be invoked with one program line.

mod date — The date a file was last written to.

mod flag — A "+" sign placed after a filename that indicates it was written to since its last backup.

MSB — The Most Significant Byte. See LSB.

NIL — A "dummy" device which a logical device can be linked or routed to. When you reset a user-defined device, it points at NIL. NIL discards any data that is sent to it and returns a null (ASCII 0) when data is requested from it. It is useful when you want to discard output from a program during a test run.

NRN — Next Record Number.

parameter — an optional value that you supply to a command line. Parameters may follow a command or utility and are enclosed in parentheses ( ).

parse — The process of breaking a command into individual parameters.

partspec — A way to represent a group of one or more files by entering only part of the file specification. Partspecs are allowed in some TRSDOS library and utility commands so that a group of files can be specified. A partspec can consist of any combination of the four fields that make up a filespec.

In a partspec, a dollar sign ($) can be used to represent any character in a given position in a filespec. This is called "wildcarding."

By prefixing a partspec with a minus sign (-), you can cause all the files *except* those that match the partspec to be considered in the given command.

.password — The optional password associated with a filespec. A password's first character is a period (.) and it is followed by one to eight alphanumeric characters, the first of which must be a letter.

PATCH — A utility that makes minor alterations to disk files.

***PR** — The Line Printer device.

RAM — Random Access Memory. This type of memory can be accessed in any order, and any byte can be read or written at any time.

ROM — Read Only Memory. This type of memory stores information that will not change. ROM does not require power to maintain its data.

RS-232-C — A standard developed by the EIA (Electronic Industries Association) that allows computer equipment made by different manufacturers to communicate with one another in a serial fashion. The standard describes the timing, voltages, connector types, and pin assignments that must be used to be called "RS-232-C."

sector — A contiguous 256-byte block of disk storage. Each sector has an I.D. field which contains its track and sector number. This allows the hardware to use the proper area of the disk when reading or writing to the disk.

A sector is the smallest amount of data the operating system will read from or write to a disk. Several sectors make up a track. One or more tracks make up a cylinder.

***SI** — The Standard Input device. Programs that read data from this device normally receive data from the keyboard. You can change this to have data read from a file or another device by issuing a ROUTE command. This allows a program to accept input from any device without the need to modify the program.

***SO** — The Standard Output Device. Data that is output to this device by a program is normally displayed on the screen. You can change this to have the data written to a file or another device by issuing a ROUTE command. This allows a program to perform output to any device without the need to change the program.

switch — A parameter with a definite setting, such as ON/OFF or YES/NO.

token — A variable used in JCL.

track — A group of sectors that are the same radius from the center of the disk.

utility — A program that provides a service to the user. Utilities differ from library commands as they are usually larger programs and require memory that is usually reserved for the user.

word — A 16-bit value which is stored in two contiguous 8-bit bytes. A word may be specified in hexadecimal format X'nnnn' or in decimal format nnnnn where nnnnn is a value from 0 to 65535.

Appendix I/TRSDOS Programs

This appendix contains five TRSDOS programs that you can use with the SET, SYSTEM, and FILTER library commands. There is a short explanation and examples for each program.

JOBLOG

ROUTE *JL [TO] *filespec*
ROUTE *JL [TO] *devspec*

Establishes the TRSDOS Joblog device (*JL), which collects certain information and sends it to a *filespec* or *devspec*.

You can use JOBLOG to create a file that contains a list of commands that you issue.

The information sent to *filespec* or *devspec* consists of all commands entered or received and the time (according to the system clock) that the commands occur.

When you issue a RESET *JL command, the Joblog function ceases and *filespec* closes. See the ROUTE library command for additional information.

To view the contents of a Joblog file, issue a RESET *JL command to close the file, and then a LIST command to list the file's contents.

To view the contents of a Joblog disk file when it is open, add a "trailing exclamation point" (!) to *filespec* (see "filespec" in the GLOSSARY). Then use the LIST library command to list the file to the screen or printer.

NOTE: If *filespec* already exists, information sent to it is appended to the end of the file.

Examples

```
ROUTE *JL TO LISTER/JBL (ENTER)
```

sends a log of all commands entered and received to the file LISTER/JBL.

```
ROUTE *JL TO *PR (ENTER)
```

sends a log of all commands entered and received to the printer.

KSM/FLT

Filter

SET *devspec* [TO] KSM/FLT [USING] *filespec* [(*parameter*)]
FILTER *KI *devspec*

Establishes the KSM (Key Stroke Multiply) filter.

You can use KSM/FLT to assign repetitive tasks (such as issuing a TRSDOS command) to one key, so that you only have to press CLEAR and the assigned key to execute the task.

devspec is any user-created devspec.

filespec contains up to 26 "key equivalents." The KSM filter loads the key equivalents from *filespec* into high memory.

The *parameter* is:

ENTER=*value* specifies *value* as the character TRSDOS recognizes as an ENTER character in a KSM file. *value* is a number in the hexadecimal format X'*nn*', a decimal number, or a single character such as a colon (:). The default *value* is the semicolon (;).

Each key equivalent is associated with the CLEAR key and an alphabetic key. When you press CLEAR and a key, TRSDOS executes the phrase associated with that key.

Building a KSM File

You can use the BUILD library command to build a /KSM file. To build a KSM file named ROUTINE/KSM, type:

```
BUILD ROUTINE/KSM (ENTER)
```

TRSDOS then lets you enter up to 26 key equivalents with the prompts:

```
A=>
B=>
C=>
.
.
.
Z=>
```

To assign a character, type in the desired command; then terminate the line by pressing (ENTER) (or the character you specified with the ENTER parameter). To skip a character, press (ENTER) at the prompt. Pressing (ENTER) does not place an (ENTER) character at the end of the key equivalent, but merely terminates your input for that key. To place an (ENTER) character in a key equivalent, type a semicolon (;) where you wish an (ENTER) press to be executed. Each line can be up to 255 characters long.

When you have assigned all 26 characters, the file is closed and the BUILD terminates. Pressing (CTRL)(SHIFT)(@) terminates the BUILD at any time.

If you want to create characters or strings that are not available from the keyboard, use the (HEX) parameter of the BUILD command.

It is not absolutely necessary to use the BUILD command with the /KSM extension to create a KSM file. The KSM/FLT program can use any file in ASCII format. TRSDOS uses the same rules concerning (ENTER) and the semicolon for a file in an ASCII format.

If you wish to deactivate the KSM filter, issue the command:

```
RESET *KI (ENTER)
```

If you wish to change to a different KSM file, issue the commands:

```
RESET *KI (ENTER)
RESET devspec (ENTER)
```

And re-issue the commands for the new file:

```
SET devspec KSM/FLT [USING] filespec (ENTER)
FILTER *KI devspec (ENTER)
```

Examples

```
A=>DIR :0 (ENTER)
```

specifies the key equivalent of A as "DIR :0". The command DIR :0 is displayed on the screen when the (CLEAR) and (A) keys are pressed together. The command is not executed until you press the (ENTER) key.

```
B=>FREE; (ENTER)
```

specifies the key equivalent of B as "FREE;". A semicolon in a key equivalent represents an (ENTER) character. So, when you press (CLEAR) and (B), the FREE library command is executed immediately (since the last character of the phrase is a semicolon).

```
F=>FREE;DEVICE; (ENTER)
```

specifies the key equivalent of F as "FREE;DEVICE;". A semicolon in a key equivalent represents an (ENTER) character. So, when you press

(CLEAR) and (B), the FREE and DEVICE library commands are executed immediately.

Error Conditions

If you attempt to use KSM/FLT with more than one device by issuing another SET command, TRSDOS displays a "Filter already attached to *xx" error message. *xx indicates the device specified in the first SET command.

When you install more than one KSM file, the second KSM file cannot be larger than the first KSM file. If the second file is larger, TRSDOS displays a "Request exceeds available memory" error message.

COM/DVR

SET *CL [TO] COM/DVR	**Driver**

In order to use the Communications Line device (*CL), you must SET it to this driver program.

You can use COM/DVR to prepare the Communications Line (*CL) for use.

COM/DVR sets *CL to the RS-232-C hardware.

After you SET *CL to the RS-232-C hardware, you can alter the parameters of the RS-232-C port with the SETCOM command.

Example

```
SET *CL TO COM/DVR
```

sets *CL to its driver program.

```
SETCOM (WORD=8,PARITY=OFF) (ENTER)
```

configures the RS-232-C port using the values specified.

Technical Information

When you set the COM/DVR, it will be placed in high memory if there is not enough room in low memory. (Low memory is within TRSDOS and does not take away from the memory available for your

programs.) If this happens, a message similar to the following appears:

```
Note: driver installed in high memory.
```

If you want to use Memdisk while you are using COM/DVR, be sure to install Memdisk first.

If you attempt to use COM/DVR before you install *CL in memory with the SET command, TRSDOS displays a "Must install via SET" error message.

If an application program sets Bit 0 of CFLAG$, TRSDOS displays the error message "No memory space available" when you attempt to load *CL. The application program must reset the bit in CFLAG$. See the *Model 4/4P Technical Reference Manual* for additional information on CFLAG$.

FORMS/FLT

Filter

SET *FF [TO] FORMS/FLT
FILTER *PR *FF

You can use FORMS/FLT to prepare the printer filter (*FF) for use.

In order to use the printer filter (*FF), you must SET it to this filter program, and activate it with the FILTER command.

After you SET *FF to FORMS/FLT, you can set up the parameters of the printer filter with the FORMS command.

Example

```
SET *FF TO FORMS/FLT (ENTER)
```

sets *FF to its filter program.

```
FILTER *PR *FF
```

filters the printer to the printer filter program.

```
FORMS (MARGIN=12,CHARS=70,INDENT=17) (ENTER)
```

configures the printer filter by causing all lines to start 12 spaces in from the normal left-hand starting position. Any line longer than 70 characters is indented 17 spaces (5 spaces past the margin) when wrapped around, so it is printed starting at position 7.

Technical Information

When you set the FORMS/FLT, it will be placed in high memory if there is not enough room in low memory. (Low memory is within TRSDOS and does not take away from the memory available for your programs.) If this happens, a message similar to the following appears:

Note: filter installed in high memory.

If you want to use Memdisk while you are using FORMS/FLT, be sure to install Memdisk first.

Error Conditions

If you attempt to use FORMS/FLT before you install *FF in memory with the SET command, TRSDOS displays a "Must install via SET" error message.

If an application program sets Bit 0 of CFLAG$, TRSDOS displays the error message "No memory space available" when you attempt to load *FF. The application program must reset the bit in CFLAG$. See the *Model 4/4P Technical Reference Manual* for additional information on CFLAG$.

If you attempt to use FORMS/FLT with more than one device by issuing another SET command, TRSDOS dislays a "Filter already attached to *xx" error message. *xx indicates the device specified in the first SET command.

MEMDISK/DCT

Driver

SYSTEM (DRIVE=*drive*,DRIVER="MEMDISK")

Lets you add a pseudo floppy drive to the system which keeps its files in memory. Files stored on this drive can be accessed, read, and written more rapidly than files on a floppy. Only one Memdisk can be installed at a time.

All TRSDOS utilities treat the Memdisk drive as any other drive, so you can COPY, BACKUP, REMOVE, PURGE, ATTRIB, and display the DIRectory of the files on the Memdisk.

drive is the drive number you wish Memdisk to be. If you specify a drive number that is already defined, it is disabled and the Memdisk takes its place. *drive* is a number from 1 to 7.

To Install the Memdisk

When you start Memdisk, the following menu is displayed:

```
<A>  Bank 0 (Primary Memory)
<B>  Bank 1
<C>  Bank 2
<D>  Banks 1 and 2
<E>  Disable MemDISK

Which type of allocation -
<A>, <B>, <C>, <D>, or <E>?
```

Each bank contains 32K of memory. If your system has only 64K of memory, then you do not have Banks 1 and 2.

Bank 0 is the top half of user memory. (See the Memory Map in the *Technical Reference Manual.*) It is shared by programs, drivers, filters, and Memdisk.

Because it is shared, if you select Bank 0 you are prompted for the number of cylinders that are to be used for the Memdisk in Bank 0. Selecting the number of cylinders allows you to use Memdisk but still have enough memory for the programs you want to run. You must select at least 3 cylinders. If you format Memdisk, the amount of memory used by each cylinder is shown below:

Double Density $256 \times 18 = 4608$ bytes per cylinder (4.5K)
Single Density $256 \times 10 = 2560$ bytes per cylinder (2.5K)

If you specify Banks 1 or 2, then all of the bank (32K) is used. If you specify option (D), then Memdisk uses Banks 1 and 2 (64K).

After selecting which bank you want to use, you see:

```
Single or Double Density <S,D>?
```

This allows you to adjust the way memory is formatted. You get the same amount of space at single density as you do at double density, but the number of sectors per cylinder differs.

This feature allows mirror image backups to be performed, which allows data to be loaded into and out of Memdisk much faster.

Memdisk looks exactly like a floppy disk to a program.

If you selected Bank 0 (Double Density), the following message is displayed:

```
Note: Each Cylinder equals 4.50K of space.
Number of free cylinders 1-N ?
```

N can be from 1 to 12. The value of N varies according to the number of other drivers resident in memory.

If you specified Bank 0 (Single Density), the following message is displayed:

```
Note:  Each Cylinder equals 2.50K of space.
Number of free cylinders 3-N ?
```

Enter the number of cylinders you want Memdisk to use in Bank 0, using the formula on the previous page. N can be from 3 to 7.

After you enter the configuring information, the following prompt is displayed:

```
Do you wish to format it <Y,N>?
```

If you have not used Memdisk before, press (Y). Formatting is not optional upon initial installation. MEMDISK is not initially installed unless you format it.

If you have used Memdisk and the system failed for some reason, press (N) to retrieve files that were left in Memdisk when it was last used. Remember that if the power went off, the Memdisk contents were erased.

If you answered the format question with (Y), you see the message:

```
Verifying RAM Cylinder NN

Verifying Complete, RAM good
Directory has been placed on Cylinder 1

MemDISK Successfully Installed
```

At this point, the Memdisk has been added to your system. The disk name is MEMDISK. It can be treated just like a floppy disk drive until you disable it or you reset the system.

To Disable the Memdisk

If you want to disable the Memdisk, then you must issue the command:

SYSTEM (DRIVE=*drive*,DRIVER="MEMDISK") (ENTER)

Then, at the menu select the (E) option. Memdisk displays one of the following messages:

```
MemDISK disabled, memory now available
MemDISK disabled, Unable to reclaim high memory
MemDISK disabled, Unable to reclaim driver area
MemDISK disabled, Unable to reclaim high memory
     and driver area
```

If you receive the first message, Memdisk was disabled and was able to reclaim all memory (driver area, high memory (Bank 0), and alternate memory banks 1 and 2) that it was using.

If you receive the second message, Memdisk was unable to reclaim high memory (Bank 0) because another driver or filter was installed after Memdisk was set up and the other program is still in the way. This is known as memory fragmentation. If you need to use this area of memory, then you must reset the system.

If you receive the third message, Memdisk was disabled and able to claim high memory or alternate bank memory, but it could not reclaim the driver area.

If you receive the fourth message, Memdisk was disabled, but it could not reclaim any memory.

Error Conditions

Memdisk should be installed *before* COM/DVR or FORMS/FLT are. Filters and drivers can be loaded into an area within TRSDOS called low memory. (This area does not take away from the memory available for your programs.) However, not all of the drivers and filters can fit into this area at the same time. If there is no room left in low memory, most of the drivers and filters can be loaded in high memory. Since low memory works on a first come, first served basis and Memdisk is the only driver or filter that *must* load into low memory, you should install Memdisk before the other drivers and filters. This ensures that there is space available in low memory for Memdisk to reside.

If you omit the DRIVE= parameter in the system command when installing Memdisk, TRSDOS displays "Logical drive number required" error message.

You cannot specify Drive 0 when installing Memdisk. You must install Memdisk as another drive, then use the SYSTEM (SYSTEM=0) command to change Drive 0 to Memdisk.

If you attempt to use Memdisk before you install it in memory with the SYSTEM command, TRSDOS displays a "Must install via SYSTEM (DRIVER=)" error message.

If you attempt to re-install Memdisk in a different area of memory than the area that it was originally installed in, you get the error message "MemDISK already Active." Memdisk must always be re-installed as it was initially installed.

If you specify the wrong drive number (in the SYSTEM (DRIVE=*drive*,DRIVER="MEMDISK") command) and you attempt to disable the Memdisk, then you receive the error message "Target Drive not a MemDISK."

If you attempt to disable a Memdisk and there is no MemDISK in the system to disable, then you receive the error message "MemDISK not present."

Technical Information

A Bank 0 Memdisk and BASIC use the same area of memory (RAM). Since a Bank 0 Memdisk and BASIC use the same area of memory, we recommend that you do not use BASIC when Memdisk is resident in Bank 0.

If you are going to use Memdisk as the system drive, you must BACKUP SYS0/SYS to it before it becomes the system drive. After Memdisk becomes the system drive, you can REMOVE SYS0/SYS from the Memdisk.

FLOPPY/DCT

Driver
SYSTEM (DRIVE=*drive*[,][DISABLE][,]DRIVER="FLOPPY")

Defines a logical drive as a floppy drive.

After you assign a slot, use DISABLE to remove access to the drive. You can re-enable the drive with different settings.

Error Conditions

If you enter FLOPPY/DCT at the TRSDOS Ready prompt, TRSDOS displays a "Must install via SYSTEM (DRIVER =)" error message.

If you have already defined a drive with the SYSTEM command, you must disable the drive with the DISABLE option before you redefine the device.

If you omit *drive* or specify an invalid drive number, TRSDOS displays a "Logical drive number required" error message.

CLICK/FLT

Filter

SET *devspec* [TO] CLICK/FLT [(CHAR=*number*)]
FILTER *KI *devspec*

Produces a tone from the sound generator inside your computer.

Before you use CLICK/FLT, you must install it in memory with the SET command. After you install the filter, the sound generator produces a tone each time you press a key on your keyboard. This is called auditory feedback.

You can change the pitch and duration of the tone by applying a patch to the values that produce the tone. The patch is:

```
PATCH CLICK/FLT.FILTER
(D00,A0=ddpp:F00,A0=18 48) (ENTER)
```

d can be a hexadecimal value in the range 1 to FF, specifying duration. 1 produces the shortest duration and FF produces the longest duration.

p can be a hexadecimal value in the range 0 to FF, specifying the pitch of the tone. 1 produces the highest pitch and FF produces the lowest pitch.

The sound that CLICK produces when the system is running at slow speed (2MHz) is different from the sound that CLICK produces when the system is running at 4 MHz.

You can include the optional parameter CHAR=*number* to produce a tone each time CLICK/FLT encounters a specific character. *number* represents the specific character and can be a hexadecimal number in the range 1 to 255. You can abbreviate CHAR to C.

Examples

```
SET *II CLICK (C=13) (ENTER)
FILTER *PR *II (ENTER)
```

Filters all data sent to the printer through the device, *II. Each time *II encounters a carriage return, (X'0D'), CLICK produces a tone.

Technical Information

If an application program sets Bit 0 of CFLAG$, TRSDOS displays the error message "No memory space available" when you attempt to load the click filter. The application program must reset the bit in CFLAG$. See the *Model 4/4P Technical Reference Manual* for additional informaton on CFLAG$.

If you attempt to use CLICK/FLT before you install the click filter in memory with the SET command, TRSDOS displays a "Must install via SET" error message.

When you install CLICK/FLT, TRSDOS places it in high memory if there is not enough room in low memory. (Low memory is within TRSDOS and does not reduce the memory available for your programs.) If this happens, a message similar to the following appears:

```
Note: filter installed in high memory.
```

If you want to use Memdisk while you are using CLICK/FLT, you must install Memdisk first.

Immediate Execution Program

Using TRSDOS Version 6.2, you can create an Immediate Execution Program (IEP). Once you create an IEP, you can load and run it at the TRSDOS Ready prompt simply by pressing (*)(ENTER).

TRSDOS stores an IEP in the SYS13/SYS file. Because TRSDOS recognizes the program as a system file, TRSDOS includes the file when creating backups and loads the program faster. To implement an IEP use the following syntax:

COPY *filespec* SYS13/SYS.LSIDOS:*drive* (C=N) (ENTER)

filespec can be any executable (/CMD) program file. *drive* specifies the destination drive. The destination drive must contain an original SYS13/SYS file.

Example

```
COPY SCRIPSIT/CMD:1 SYS13/SYS.LSIDOS:0 (C=N) (ENTER)
```

TRSDOS copies SCRIPSIT/CMD from Drive 1 to SYS13/SYS in Drive 0. At the TRSDOS Ready prompt, when you press (*)(ENTER), TRSDOS executes SCRIPSIT.

Error Conditions

If you type (*)(ENTER) before you copy a file to the SYS13/SYS file, TRSDOS displays a "No command <*> present. as SYS13" error message.

Appendix J/ Memory Maps

TRSDOS Memory Map

0000H to 25FFH	Operating System	Reserved for TRSDOS operations.
2600H to 2FFFH	Overlay Area	Used by all library commands and some utilities. If BASIC or other programs use this area, you must not allow SYSTEM commands (through the @CMNDR SVC) or you must reload this area when the SVC returns.
3000H to HIGH$	User Program	Reserved for applications programs.
HIGH$ to FFFFH	Driver/Filter/User or System tasks	Reserved for system drivers, filters, and tasks. The spooler, and drivers that cannot fit into the area reserved in TRSDOS are stored in this area of memory. When MEMDISK resides in Bank 0, it is also stored in this area of memory.

HIGH$ is a pointer maintained by TRSDOS. It moves depending on the number and size of modules stored above the address that HIGH$ points at. You can use the MEMORY command to display or alter the value of HIGH$.

BASIC Memory Map

0000H to 25FFH	Operating System	Reserved for TRSDOS operations.
2600H to 2FFFH	Overlay Area	Used alternately by TRSDOS and BASIC. Whenever you use a TRSDOS library command, TRSDOS uses this area to store the program that will perform the command. BASIC reloads this area with its data when you return from TRSDOS.
3000H to 80FFH	BASIC	Reserved for BASIC.
8100H to Bottom of Stack	User's BASIC Program	Reserved for your programs, variables, strings, and arrays.
Bottom of Stack to HIGH$ or User-Defined top of memory (M)	BASIC stack and File Control Block(s)	Contains the stack used by BASIC and the File Control Block(s) (FCBs). Each FCB requires 564 bytes of storage. The number of FCBs that your system has is selected with the command: BASIC (F=n), where 'n' specifies the number of files that can be open at any one time. (One additional 564-byte block is always allocated and is reserved for use by BASIC.)
User-Defined top of memory (M) or HIGH$ to HIGH$	Assembly language routines callable from BASIC.	This area exists only if you create it with the command, BASIC (M=address) where 'address' specifies the last address that BASIC will use. The area between "M" and HIGH$ is used to store assembly language routines that are called by BASIC programs.

HIGH$ to FFFFH	Driver/Filter/User or System tasks	Area in which drivers, filters, and tasks that are continuously used by the system are stored. Items in this area include the spooler, drivers and filters that cannot fit into the area reserved within TRSDOS, and MEMDISK (when it resides in BANK 0.) Assembly language routines that are to be called from BASIC may be placed here as long as the programs follow the rules outlined in the *Model 4/4P Technical Reference Manual.*

User Program

Your User Program space is dynamic. It is dependent on the number of data files you requested when loading BASIC (called "concurrent" files), the HIGH$ marker, the amount of stack space, and the highest memory location you specified when loading BASIC. For information on how to load BASIC, see Chapter 1.

If the HIGH$ marker is at the top of physical memory (FFFFH) and you do not set the highest memory location BASIC can use (with the M parameter), there is 31 K bytes of memory available for your program. Each concurrent file uses 564 bytes of memory.

If HIGH$ is not at FFFFH, or if you specify the M parameter when loading BASIC, use the PRINT FRE(0) command to display the amount of program space available.

The number of concurrent files also determine the location of the top of the stack. BASIC uses this formula:

location = M − (564 × *number of concurrent files*) − 564

Location is the top of the stack. You can use the CLEAR statement to set aside additional stack space. Additional stack space decreases the amount of program space available.

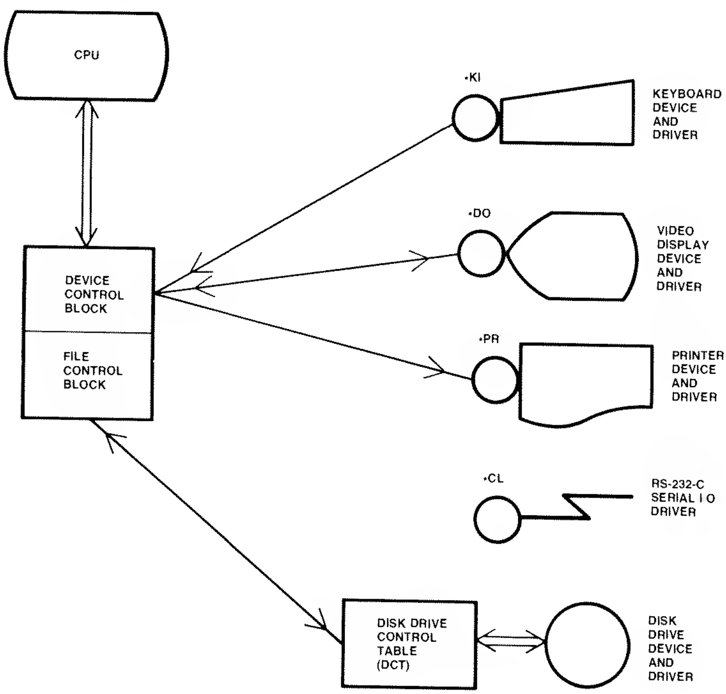
Appendix K/ Using The Device-Related Commands

The advanced, device-related commands affect the assigned TRSDOS devices and the devices that you create. They are:

DEVICE, FILTER, LINK, SET, ROUTE, RESET

DEVICE is different from the other commands because instead of directly affecting the devices, DEVICE actually shows how each device is set up and what connections between devices (and files) exist. So, each time you issue one of the above commands, you should issue a DEVICE (B) command to make sure the devices are set the way you want them.

This figure illustrates the device structure standard configuration, that is, how the devices are routed at their normal start-up state under TRSDOS 6. The illustrations on the following pages illustrate the system device configuration after certain commands are executed. We assume that before each command or group of commands is executed, that the device configuration is in its normal start-up state as illustrated in this flowchart.



Creating an Unfiltered Link

An unfiltered link is different from a filtered link because there is not an in-between program (a filter) that affects the data flowing between the two devices.

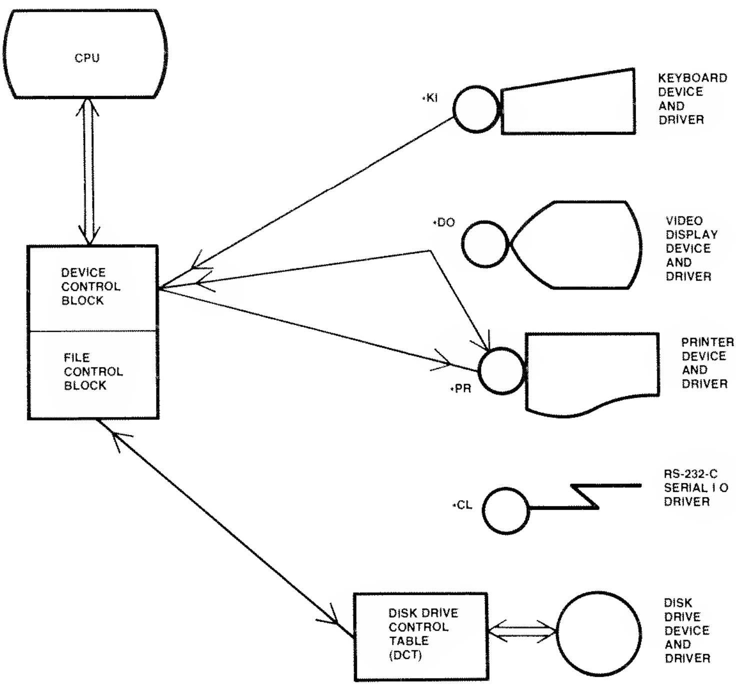
Creating an unfiltered link between a device and a file involves the ROUTE and LINK commands.

ROUTE can create a user device and routes it to a file.

The command

```
ROUTE *DO TO *PR (ENTER)
```

redirects data sent to the video display to the printer. All data that prints on the screen prints on the printer as well. After executing this statement, the device configuration structure is:



LINK creates a link between two devices.

Remember that it is a good idea to issue a DEVICE command before you create a link. In the following example, we are going to route the printer. On start-up, the printer is shown in the device table as:

```
*PR => X'0E0F'
```

The device table entry shows the place in memory (X'0E0F') where the driver program that controls the printer is located. This memory address may vary.

Example

In this example we are going to link the printer to a file. That is, all data sent to the printer is also sent to the file.

To create a link between the printer (*PR) and the file PRINT/TXT:0:

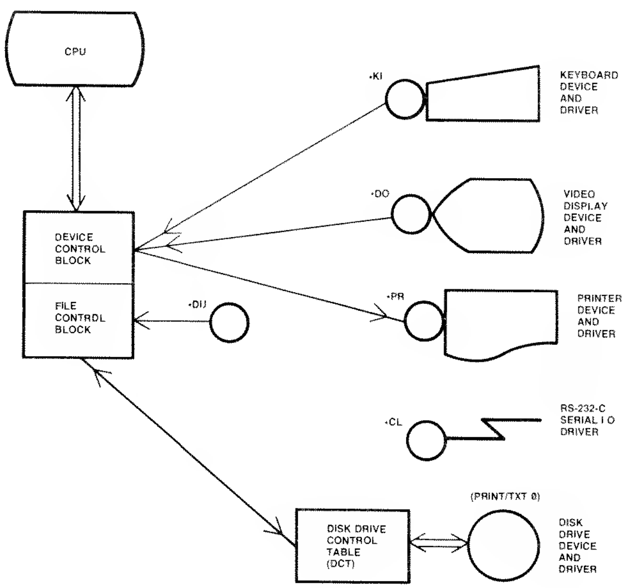
1. Create a logical device *DU, open the file PRINT/TXT on Drive 0, and route the device to the file by issuing the command:

   ```
   ROUTE *DU TO PRINT/TXT:0 (ENTER)
   ```

   The device table shows:

   ```
   *DU <=> PRINT/TXT:0
   ```

   The device configuration is:



Everything that TRSDOS sends to *DU is sent to the file PRINT/TXT on Drive 0.

2. Link the printer to *DU, which in turn is routed to PRINT/TXT by issuing the command:

```
LINK *PR *DU (ENTER)
```

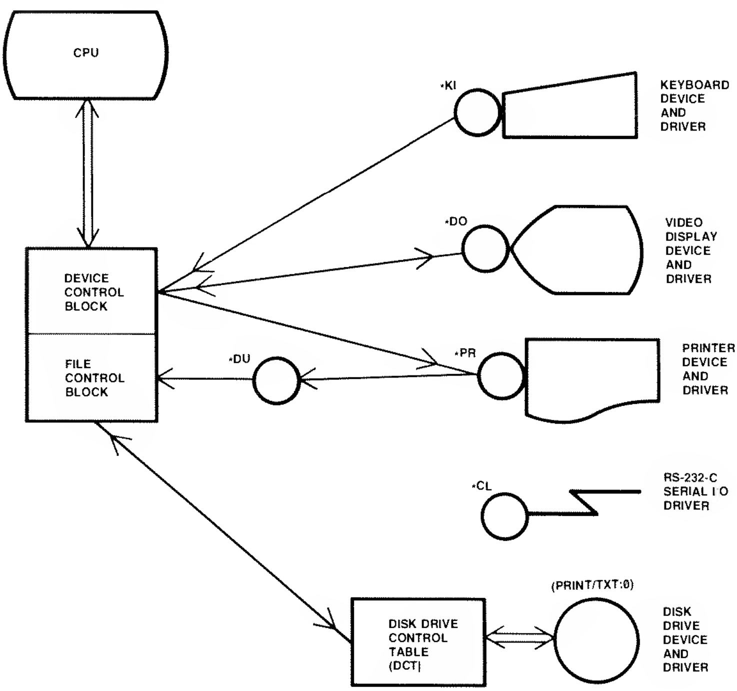
the device table shows:

```
*PR => *L0 : *DU & => X'0E0F'
*DU <=> PRINT/TXT:0
```

The following link now exists:

*PR → Printer Driver (at X'0E0F')
*PR → *DU ↔ PRINT/TXT

The device configuration is:



Everything that TRSDOS sends to *PR is also sent to *DU and from there to PRINT/TXT on Drive 0.

Creating a Filtered Link

Creating a filtered link involves the SET and FILTER commands. A filtered link involves a devices and a filter program which affects the data that flows to or from the device.

SET prepares a user-created device for the filter connection.

FILTER creates the "logical link" between two devices. The first device is usually a system device, and the second device is always a user-created filter device.

Example

To create a filter link you need a filter program. In this example we use the system filter program FORMS/FLT.

Before you issue a SET or FILTER command, be sure to issue a DEVICE command to see the start-up conditions of the system devices. In this example, we are going to filter the printer device. On start-up, the printer is shown in the device table as:

```
*PR <= X'0E0F'
```

The device table entry shows the place in memory (X'0E0F') where the driver program that controls the printer is located. This memory address may vary.

To create a FORMS filter link between *PR and a logical device *FF.

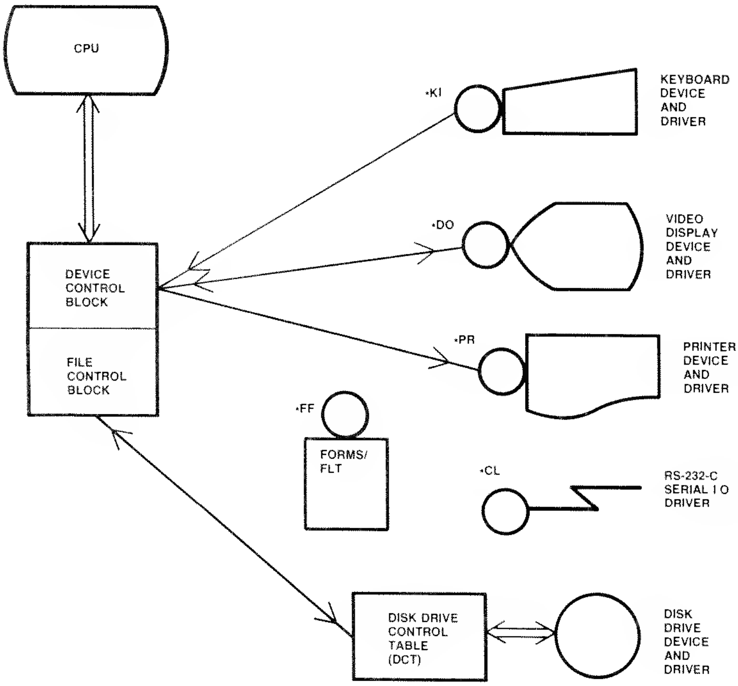
1. Set *FF to the FORMS filter by issuing the command:

   ```
   SET *FF TO FORMS/FLT (ENTER)
   ```

   The device table shows:

   ```
   *PR <= X'0E0F'
   *FF <# [Inactive] X'FF67'
   Options: Type, Forms
   ```

The device configuration is:



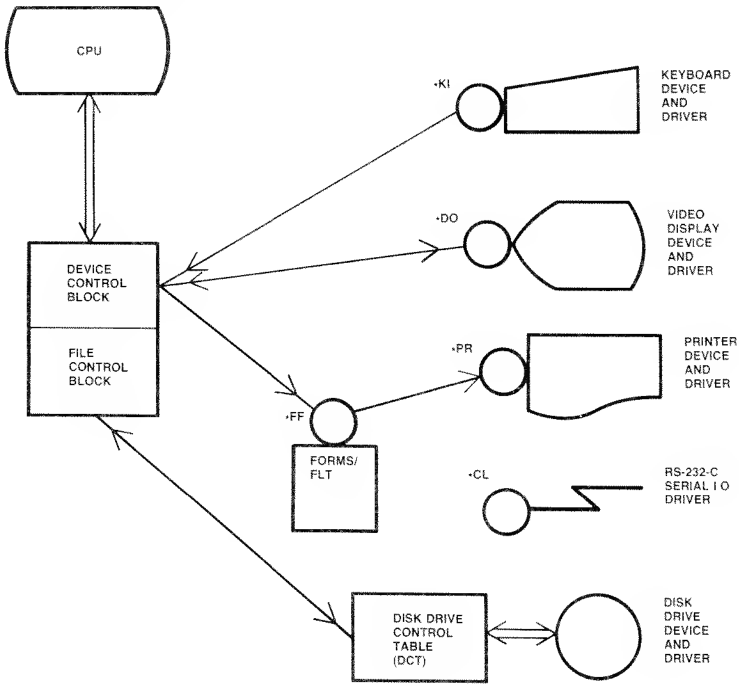
2. Now use the FILTER command to connect the FORMS filter program to the printer by issuing the command:

```
FILTER *PR TO *FF (ENTER)
```

The device table shows:

```
*PR <# [*FF] X'FF67'
*FF <= X'0E0F'
Options: Type, Forms
```

The device configuration is:



The following link now exists:

Printer Driver → *FF → FORMS/FLT → *PR

Everything that goes to the printer is sent through *FF, filtered through the FORMS filter and printed on the printer device.

Using the RESET Command

You can use RESET with SET, FILTER, ROUTE, or LINK. In this example, we show you what happens when you break the link between —PR and PRINT/TXT.

Example

To break the link:

*PR → Printer Driver (at X'0E0F')
*PR → *DU ←→ PRINT/TXT

1. First, to remove the routing between *DU and PRINT/TXT, issue the command:

```
RESET *DU (ENTER)
```

The device table shows:

```
*PR -> *L0 | *DU & => X'0E0F'
*DU <=> NIL
```

The following link now exists:

*PR → Printer Driver (at X'0E0F')
*PR → *DU ←→ NIL

All output sent to *PR is still sent to *DU, even though *DU is pointed NIL.

2. To remove the link between *PR and *DU, issue the command:

```
RESET *PR (ENTER)
```

The device table shows:

```
*PR => X'0E0F'
*DU <=> NIL
```

Now you have returned *PR to its original start-up condition, and the link between *PR and *DU no longer exists.

(You can type REMOVE *DU to remove *DU from the device table.)

Appendix L/ Setup for 50 Hz AC Power (non-USA users)

If you have had your computer adjusted to operate in areas where the AC power is 50 Hertz (Hz) rather than 60 Hz, you can execute a SYSTEM (HERTZ) command. This command applies a patch to your TRSDOS system diskette to make the software aware of the fact that the hardware is operating at 50 Hz.

If you do not apply this patch, the time-of-day clock runs at the wrong speed. Failure to apply this patch causes no other problems. **Your hardware must have been modified before you apply this patch.**

Before you execute a SYSTEM (HERTZ) command, make a backup of your master diskette. Remove your master diskette from Drive 0 and insert the backup diskette in Drive 0. At the TRSDOS Ready prompt, type:

```
SYSTEM (HERTZ5) (ENTER)
```

If you have your hardware readjusted to operate at 60 Hz rather than 50 Hz, you can reverse this procedure. At the TRSDOS Ready prompt, type:

```
SYSTEM (HERTZ6) (ENTER)
```

After you execute a SYSTEM HERTZ command, TRSDOS displays the TRSDOS Ready prompt. TRSDOS stores the Hertz value on the diskette in Drive 0. Press RESET to put the change into effect. This loads the patched software into RAM.

Appendix M/ Backup Limited Diskettes

Some software products distributed by Tandy come on backup limited diskettes. This means that you can make only a fixed number of copies of the master diskette that you receive. You should use the master diskette to make only the backup copies that you will use, as you cannot make a backup copy of a backup that was made from a backup limited diskette.

These diskettes are clearly marked to indicate that they are backup limited. If you are uncertain, contact your Radio Shack Computer Center™ or the store where you purchased the diskette.

When you have exhausted the number of copies you are allowed to make or if the master diskette is write protected, the following message appears when you attempt to back up the diskette:

```
Protected source disk
```

Making a Backup Copy

Before you make a copy of a backup limited diskette, you must remove the write-protect tab from the diskette (if one is present). Because the diskette is not write-protected, you should be very careful that you do not accidentally format the master diskette or back up the blank diskette to the master diskette.

Follow the steps given below:

1. Insert a TRSDOS system diskette into floppy Drive 0. Insert a blank diskette into floppy Drive 1.
2. Format the blank diskette, following the directions given with the FORMAT utility. (You can use the command FORMAT :1 (Q=N) (ENTER) to produce a default diskette.)

   If the diskette has any flaws on it (that is, if an asterisk is displayed next to one or more cylinder numbers), repeat step 2 with another blank diskette. Remember that you can make only a fixed number of copies of this diskette, so you should try to use good media.
3. At TRSDOS Ready, type:

   ```
   BACKUP :0 :1 (X) (ENTER)
   ```
4. When you see the prompt:

   ```
   Insert SOURCE disk &lt;ENTER&gt;
   ```

remove the TRSDOS system diskette from Drive 0 and set it aside.

Remove the write-protect tab (if any) from the master backup limited diskette you want to copy. This will be the SOURCE diskette.

Place the backup limited diskette in Drive 0 and press (ENTER).

5. The following message may appear:

```
Destination disk ID is different:
  Name=diskname  Date=mm/dd/yy
Are you sure you want to backup to
  it <Y,N> ?
```

   Respond by typing (Y) (ENTER)

6. The computer now performs the backup. When you see the prompt:

```
Insert SYSTEM disk <ENTER>
```

   remove the backup limited diskette from Drive 0 and place a write-protect tab on it.

   Insert the TRSDOS system diskette in Drive 0 and press (ENTER). A message is displayed telling you if the backup operation was successful or not. If there was an error, start over with step 1 using another blank diskette. Unsuccessful backups do not count against the number of backups you can make.

   Remove the new backup copy from Drive 1. Place a write-protect tab on it and place a label on the jacket to identify it.

Backing up selected files

You can move the programs on a backup limited diskette to the hard disk using backup by class or backup reconstruct. (The latter occurs automatically when the target drive is a hard disk.) This is counted the same as making a diskette copy using the procedure described above.

Note that if you do a backup by class and move only selected files, and if any of the files that are moved are protected, it is counted as though you made a copy of the entire disk. For example, suppose that you are allowed to make three backups of a backup limited diskette. You do a backup by class to move visible files. If one of the visible files is protected, then that file is copied along with the other visible files. However, you can now make only two more copies of the files on the master disk.

For this reason, you should be careful that you do not cheat yourself out of a copy. When moving files to the hard disk from a backup limited diskette, ask for all of the files using the (SYS,INV) options in the BACKUP command. If this moves some unwanted material, it can be purged later.

You may use backup by class or backup reconstruct to move non-protected files to and from the hard disk or a backup of the backup limited diskette. However, the protected files are not backed up and will not be listed if you use the QUERY option.

Appendix N/ Converting Model 4 Data Files to Model III Mode

This appendix describes how you can use data files created on your computer (under TRSDOS Version 6) in Model III mode.

Start up your system under TRSDOS Version 6. At the TRSDOS Ready prompt, insert a blank, write-enabled diskette into Drive 1. Format the Drive 1 diskette by typing:

```
FORMAT :1 (SDEN,CYL=35,Q=N,ABS) (ENTER)
```

This command makes the diskette almost identical to those formatted by Model I disk operating system (TRSDOS Version 2.3). Later, you will copy the files from this diskette to a TRSDOS Version 1.3 diskette.

Use the COPY command with the CLONE = NO Option, to copy files from Drive 0 to Drive 1. If any of the files have a password, use the ATTRIB command to remove the password.

Remove the Version 6 diskette from Drive 0 and insert a TRSDOS Version 1.3 diskette. Press RESET to startup your system under Version 1.3.

After you enter the date and time, type:

```
CONVERT (ENTER)
```

When CONVERT asks for the source drive, type (1) (ENTER). When it asks for the destination drive, type (0) (ENTER).

CONVERT reads the diskette in Drive 1. If a file of the same name already exists on the Drive 0 diskette, CONVERT asks you if you want to copy over the file. If you respond by typing (Y) (ENTER), the file on the disk in Drive 1 replaces the file on the diskette in Drive 0 and TRSDOS moves all files on the Drive 1 diskette to Drive 0. If you respond (N) (ENTER), TRSDOS copies all other files except that one.

When CONVERT is complete, the TRSDOS Ready prompt appears. You can now reuse the diskette in Drive 1.

Index

CUSTOM MANUFACTURED FOR RADIO SHACK,
A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
91 KURRAJONG AVENUE MOUNT DRUITT, N.S.W. 2770	PARC INDUSTRIEL 5140 NANINNE (NAMUR)	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

7/85-TM

874-9747

Printed in U.S.A.